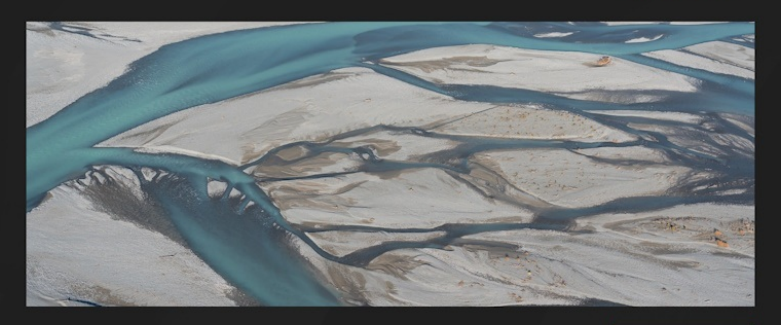
<packt>



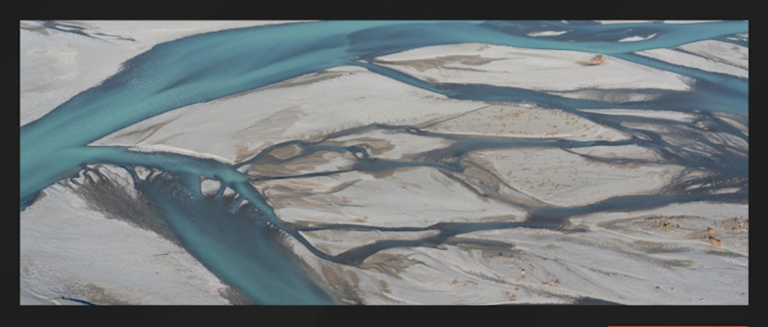
1-Е ИЗДАНИЕ

Изучаем GDScript разрабатывая игру с помощью Godot 4

Увлекательное введение в программирование на GDScript 2.0 и разработку игр с использованием движка Godot

САНДЕР ВАНХОВ

<packt>



1ST EDITION

Learning GDScript by Developing a Game with Godot 4

A fun introduction to programming in GDScript 2.0 and game development using the Godot Engine

SANDER VANHOVE

Изучение GDScript путём разработки игры с помощью Godot 4

Увлекательное введение в программирование на GDScript 2.0 и разработку игр с использованием движка Godot

Сандер Ванхов



Изучение GDScript путём разработки игры с помощью Godot 4

При подготовке этой книги были приложены все усилия для обеспечения точности представленной информации. Однако

Group Product Manager: Rohit Rajkumar

Publishing Product Manager: Kaustubh Manglurkar

Book Project Manager: Sonam Pandey

Senior Editor: Anuradha Joglekar

Technical Editor: K Bimala Singha

Copy Editor: Safis Editing

Proofreader: Anuradha Joglekar

Indexer: Subalakshmi Govindhan

Production Designer: Gokul Raj S.T

DevRel Marketing Coordinators: Anamika Singh and Nivedita Pandey

Первая публикация: май 2024 г.

Production reference: 1170424

Published by

Packt Publishing Ltd.

Grosvenor House

11 St Paul’s Square

Birmingham

B3 1RB, UK

ISBN 978-1-80461-698-7

www.packtpub.com

Всем новым пользователям Godot Engine желаю, чтобы ваши игры были великолепными!

- Сандер Ванхов

Участники

Об авторе

Сандер Ванхов (Sander Vanhove) — ведущий разработчик игр и технический художник в Studio Tolima, брюссельской студии, работающей над своей первой коммерческой игрой Koira.

Создание небольших игр для своих друзей с 14 лет привело его к получению степени магистра компьютерных наук, после чего он несколько лет работал аналитиком данных.

В 2018 году он решил попробовать Godot Engine и больше не оглядывался назад. Имея за плечами гору джемовых игр, бесчисленные игровые эксперименты и несколько небольших коммерческих игровых релизов, он, наконец, решился на постоянную работу в игровой студии, вышеупомянутой Studio Tolima.

Его всегда можно застать работающим над очередным проектом, изучающим новые методы разработки игр или пытающимся научить людей пользоваться движком Godot Engine.

Я хочу поблагодарить моих родителей, Мартину и Геррита, а также близких друзей, которые верили в меня и этот проект и постоянно интересовались его ходом.

О рецензентах

Джесси Гловер (Jesse Glover) — опытный консультант по программному обеспечению с обширным опытом разработки как бэкенда, так и фронтенда, специализирующийся на разработке и администрировании Salesforce и ServiceNow. Джесси отточил свои навыки в различных технических областях, уделяя особое внимание современным фронтенд-фреймворкам и инструментам.

Помимо своих профессиональных достижений, Джесси с энтузиазмом делится своими знаниями и опытом через свой канал на YouTube **GameDevMadeEasy**, где он создаёт образовательный контент по разработке игр и программного обеспечения.

Джесси также был автором книги *Unity 2018 Augmented Reality Projects*, которая доступна на сайте Packt.

Людовик де Сен-Вьянс (Ludovic de Saint-Viance), известный в интернете как *theLudovyc*, — старший разработчик, который любит использовать Godot в свободное время. Он создавал различные проекты (вы можете найти их на его страницах Itch.io или GitHub) в течение 6 лет. Также он был администратором французского Discord-сервера о Godot: Espace Godot.

Я благодарю Никиту Рагани из Packt, которая пригласила меня написать рецензию на эту книгу, и моего менеджера Сонам Панди. Я также очень благодарен автору этой книги, Сандеру, и Packt за то, что они стали частью этого невероятного путешествия. Цитируя Сандера, «Я хочу уважать правила Packt, но мне кажется странным представляться так, будто я на странице Википедии. Если кто-то дочитает эти строки до конца, то моя любимая пицца — с ананасом».

Оглавление

Предисловие

Часть 1: Изучение программирования

1

Настройка среды

Технические требования

Игровой движок Godot и программное обеспечение с открытым исходным кодом

Немного информации о движке

Что такое программное обеспечение с открытым исходным кодом?

Получение и настройка Godot

Загрузка движка

Создание нового проекта

Светлая тема

Создание главной сцены

Краткий обзор пользовательского интерфейса

Пишем наш первый скрипт

Документация движка Godot

Присоединяйтесь к нашему сообществу!

Итоги

Опрос

2

Знакомство с переменными и потоком управления

Технические требования

Что такое переменные?

Переменные – ящики в картотеке, полные данных

Именование переменных

Переменные в GDScript

Печать переменных

Изменение значения переменной

Математические операторы

Другие операторы присваивания

Типы данных – целые числа, числа с плавающей точкой и строки

Целые числа

Числа с плавающей точкой

Строки

Что такое константы?

Константы в GDScript

Магические числа

Создание новых сцен

Начало работы с потоком управления

Оператор if

Оператор if-else

Выражение elif

Комментарии в коде

Отступы

Булева логика

Выражение сопоставления (match)

Тернарный оператор if

Дополнительные упражнения – Заточка топора

Итоги

Опрос

3

Группировка информации в массивах, циклах и словарях

Технические требования

Массивы

Создание массива

Доступ к значениям

Доступ к элементам в обратном направлении

Изменение элементов массива

Типы данных в массивах

Строки — это тайные массивы

Манипулирование массивами

Не бойтесь ошибок и предупреждений

Циклы

Циклы For

Циклы While

Продолжение или прерывание цикла

Словари

Создание словаря

Типы данных в словарях

Доступ к значениям и их изменение

Создание новой пары ключ-значение

Полезные функции

Обход словарей

Вложенные циклы

Null

Дополнительные упражнения – Заточка топора

Итоги

Опрос

4

Создание структуры с помощью методов и классов

Технические требования

Методы — это фрагменты кода, которые можно использовать повторно

Что такое функция?

Определение функции

Именование функций

Ключевое слово return

Ключевое слово pass

Необязательные параметры

Классы группируют код и данные вместе

Определение класса

Создание экземпляра класса

Именование классов

Расширение класса

Каждый скрипт — класс!

Когда доступны определённые переменные?

Область действия функции

Типы помогают нам узнать, как использовать переменную

Что такое подсказка типа?

Подсказки типа переменных

Подсказки типа массивов

Изучение типа Variant

Подсказки типа параметров функции

Подсказки типа возвращаемых значений функции

Использование void в качестве возвращаемого значения функции

Выведенные типы

null может быть любого типа

Автодополнение

Использование подсказок типов для именованных классов

Производительность

Редактор добавляет подсказки типа

Учебник по ООП

Наследование

Абстракция

Инкапсуляция

Полиморфизм

Дополнительные упражнения – Заточка топора

Итоги

Опрос

5

Как и почему следует поддерживать чистоту кода

Технические требования

Возвращаемся к именованию вещей

Соглашения об именовании

Общие советы по именованию

Публичные и приватные члены класса

Создавайте короткие функции

Не повторяйтесь (DRY)

Делайте вещи проще (KISS)

Защищёное программирование

Руководства по стилю программирования

Пробелы

Пустые строки

Длина строки

Использование документации

Доступ к документации класса

Прямой доступ к документации функций или переменных

Переходим к определению функции или переменной

Поиск документации

Доступ к онлайн-документации

Итоги

Опрос

Часть 2: Создание игры в Godot Engine

6

Создание собственного мира в Godot

Технические требования

Игровой дизайн

Жанр

Механика

История

Создание игрового персонажа

Добавление спрайта

Отображение здоровья

Манипулирование узлами в редакторе

Создание скрипта игрока

Ссылки на узлы в скрипте

Кэширование ссылок на узлы

Тестируем скрипт игрока

Экспорт переменных в редактор

Сеттеры и геттеры

Изменение значений во время игры

Различные типы экспортируемых переменных

Создание маленького мира

Изменение цвета фона

Добавление валунов Polygon2D

Порядок отрисовки узлов

Создание внешней стены

Используем творческий подход

Дополнительные упражнения – Заточка топора

Итоги

Опрос

7

Заставляем персонажа двигаться

Технические требования

Освежаем знания по векторной математике

Двумерная система координат

Что такое вектор?

Масштабирование векторов

Сложение и вычитание векторов

Другие векторные операции

Перемещение персонажа игрока

Изменение текущего узла игрока

Приложение сил к игроку

Функции _process и _physics process

Список действий

Использование ввода

Сглаживание движения

Отладка запущенной игры

Точки останова

Удалённое дерево

Дополнительные упражнения – Заточка топора

Итоги

Опрос

8

Разделение и повторное использование сцен

Технические требования

Сохранение ветки как новой сцены

Создание отдельной сцены игрока

Корневой узел сцены

Использование сохранённых сцен

Организация файлов сцен

Дополнительные упражнения – Заточка топора

Итоги

Опрос

9

Камеры, столкновения и предметы коллекционирования

Технические требования

Создание камеры, которая следует за игроком

Настройка базовой камеры

Добавление полей перетаскивания

Заставляем камеру смотреть вперёд

Сглаживаем взгляд вперед

Столкновения

Различные физические тела

Узел Area2D

Добавление формы столкновения к узлу игрока

Создание статических тел для валунов

Создание статических тел для стен

Создание предметов коллекционирования

Создание базовой сцены коллекционного предмета

Наследование от базовой сцены

Подключение к сигналу

Написание кода для предметов коллекционирования

Использование слоёв столкновений и масок

Ваш ход!

Дополнительные упражнения – Заточка топора

Итоги

Опрос

10

Создание меню, создание врагов и использование автозагрузок

Технические требования

Создание меню

Узлы управления

Создание базового меню запуска

Установка главной сцены

Создание врагов

Построение базовой сцены

Навигация врагов

Написание скрипта врага

Нанесение урона игроку при столкновении

Появление врагов и предметов коллекционирования

Создание экрана Game Over

Стрельба снарядами

Создание базовой сцены

Написание логики снаряда

Появление снарядов

Сохранение рекордов в автозагрузках

Использование автозагрузки

Создание автозагрузки HighscoreManager

Автозагрузки в удалённом дереве

Добавление пользовательского интерфейса в главное меню и игровую сцену

Использование рекорда в главном меню

Дополнительные упражнения – Заточка топора

Итоги

Опрос

11

Совместная игра в многопользовательском режиме

Технические требования

Ускоренный курс по компьютерным сетям

Что такое транспортный уровень?

Что такое уровень приложений?

Сетевое взаимодействие в Godot Engine

Изучение IP-адресов

Использование номеров портов

Настройка базового сетевого кода

Создание соединения клиент-сервер

Добавление пользовательского интерфейса

Запуск нескольких экземпляров отладки одновременно

Синхронизация разных клиентов

Обновление сцены игрока для многопользовательской игры

Синхронизация EntitySpawner

Синхронизация врага и предметов коллекционирования

Синхронизация снаряда

Исправление таймера и завершение игры

Синхронизация таймера

Синхронизация конца игры

Запуск игры на нескольких компьютерах

Отображение IP-адреса сервера

Подключение с другого компьютера

Дополнительные упражнения – Заточка топора

Итоги

Опрос

Часть 3: Углубление наших знаний

12

Экспорт на несколько платформ

Технические требования

Экспорт для Windows, Mac и Linux

Загрузка шаблона экспорта

Осуществление фактического экспорта игры

Загружаем нашу игру на Itch.io

Что такое Itch.io?

Экспортируем нашу игру для сети

Загрузка на Itch.io

Экспорт нашей игры на другие платформы

Мобильные платформы

Консоли

Итоги

Опрос

13

Продолжение ООП и продвинутые темы

Технические требования

Ключевое слово super

Статические переменные и функции

Перечисления

Лямбда-функции

Создание лямбда-функции

Где используются лямбда-функции

Передача параметров по значению или ссылке

Передача по значению

Передача по ссылке

Аннотация @tool

Итоги

Опрос

14

Расширенные шаблоны программирования

Технические требования

Что такое шаблоны программирования?

Исследуем шину событий

Проблема

Решение

Исследуем пул объектов

Проблема

Решение

Реализация пула объектов в нашей игре

Работа с конечными автоматами

Проблема

Решение

Пример состояния

Дополнительные упражнения – Заточка топора

Итоги

Опрос

15

Использование файловой системы

Технические требования

Что такое файловая система?

Пути к файлам

Путь пользователя

Создание системы сохранения

Запись данных на диск

Чтение данных с диска

Подготовка менеджера сохранений для использования в игре

Настройка игры для использования менеджера сохранений

Просмотр файла сохранения

Итоги

Опрос

16

Что дальше?

Идеи для ваших следующих проектов

Начало нового проекта

Расширение игры на выживание

Создание ещё одной игры

Бесплатные игровые ресурсы

Изучение новых тем

Следование конкретным руководствам

Читайте больше книг

Чтение документации Godot Engine

Просмотр игрового кода чужих проектов

Присоединение к сообществу

Присоединяйтесь к Форуму, Discord, Reddit или любой другой платформе

Вклад в проект Godot Engine

Присоединяйтесь к игровому джему

Завершение

Алфавитный указатель

Другие книги, которые вам могут понравиться

Предисловие

Godot Engine — самый популярный бесплатный игровой движок с открытым исходным кодом на рынке. С появлением Godot 4.0 и выпуском многих хитовых игр, созданных на Godot, таких как *Dome Keeper*, *Brotato* и *Case of the Golden Idol*, эта популярность только возросла. Сейчас самое время научиться использовать этот замечательный инструмент для разработки игр.

Изучение программирования и использования нового игрового движка может оказаться непростой задачей. Однако эта книга шаг за шагом проведёт вас через все аспекты создания собственной игры с нуля, от основ написания наших первых скриптов в GDScript до более сложных тем.

Мы научимся программировать на GDScript, пользовательском языке Godot Engine, который прост в изучении, но очень эффективен и эффективен для разработки игр. Затем мы рассмотрим все тонкости интуитивного графического интерфейса движка и узнаем всё о его гибком подходе к разработке игр на основе узлов.

Для кого эта книга

Эта книга для программистов, гейм-дизайнеров, разработчиков игр и игровых художников, которые хотят начать создавать игры в Godot 4. Если вы новичок в кодировании или разработке игр, ищете новый творческий выход и хотите попробовать Godot 4 и GDScript 2.0, эта книга для вас. Хотя никаких предварительных знаний о программировании или Godot не требуется, эта книга постепенно вводит более сложные концепции по мере продвижения по главам.

О чём эта книга

Глава 1, *Настройка среды*, начинает книгу с настройки всего

необходимого для создания игр на движке Godot Engine, а также дает краткий обзор движка и того, как писать скрипты.

Глава 2, *Знакомство с переменными и потоком управления,* объясняет основные концепции того, что такое переменные и как мы можем хранить данные в них. Затем мы рассмотрим различные потоки управления, которые помогают нам принимать решения во время выполнения нашей игры.

Глава 3, *Группировка информации в массивах, циклах и словарях*, рассказывает о двух новых типах данных: массивах и словарях. Они помогут нам группировать данные в более структурированном формате. По ходу дела мы узнаем о двух различных видах циклов, с помощью которых мы можем выполнять циклы по различным наборам данных.

Глава 4, *Структурирование с помощью методов и классов*, посвящена написанию повторно используемых фрагментов кода с использованием методов и структурированию переменных и методов в классы.

Глава 5, *Как и почему следует поддерживать чистоту кода*, знакомит нас со многими концепциями написания чистого кода, которые помогут нам создавать код, который можно использовать повторно и который будет понятен как другим, так и нам самим.

Глава 6, *Создание собственного мира в Godot*, положит начало нашему собственному игровому проекту. Мы начнем с определения того, какую игру мы будем делать, и перейдем к созданию основы игрового персонажа и среды, в которой он будет перемещаться.

Глава 7, *Заставляем персрнажа двигаться*, предлагает освежить в памяти векторную математику, которая является неотъемлемой частью перемещения объектов в двумерном пространстве. Затем мы напишем физический код, чтобы заставить нашего персонажа двигаться, и займёмся отладкой игры во время её работы.

Глава 8, *Разделение и повторное использование сцен*, показывает,

как можно легко разделить нашу игру на несколько меньших сцен, которыми легче управлять и поддерживать их, а затем рассказывается, как можно организовать все файлы сцен и скриптов в аккуратных папках внутри проекта.

Глава 9, *Камеры, столкновения и предметы коллекционирования*, начинается с создания плавной камеры, которая будет следовать за персонажем игрока, не вызывая тошноту у реального игрока. После этого мы перейдём к обработке столкновений с ландшафтом и созданию предметов коллекционирования.

Глава 10, *Создание меню, создание врагов и использование автозагрузок*, завершает нашу однопользовательскую игру, обучая нас системе меню Godot Engine, за которой следует создание врагов, которые могут перемещаться по миру, и снарядов, которыми игрок может стрелять в этих врагов. Мы завершаем эту главу введением в автозагрузку, с помощью которой мы можем сохранять рекорды.

Глава 11, *Совместная игра в многопользовательском режиме*, преобразует наш однопользовательский опыт в многопользовательский. Мы начнём с ускоренного курса по компьютерным сетям. После этого мы узнаем об узлах **MultiplayerSpawner** и **MultiplayerSynchronizer**, чтобы иметь возможность играть в нашу игру по сети с другими игроками.

Глава 12, *Экспорт на несколько платформ*, показывает, как мы можем экспортировать игру для разных платформ, таких как Windows, macOS, Linux и даже веб. Мы завершим главу, загрузив нашу игру на Itch.io, популярную платформу для инди-игр.

Глава 13, *Продолжение и дополнительные темы ООП*, знакомит с более сложными **темами объектно-ориентированного программирования (ООП)**, такими как ключевое слово **super**, статические переменные, перечисления, лямбда-функции, различные способы передачи значений методам и ключевое слово **tool**.

Глава 14, *Продвинутые шаблоны программирования*, даёт нам

основу для шаблонов программирования и исследует шаблоны шины событий, пула объектов и конечного автомата, чтобы мы могли использовать их в нашем следующем проекте.

Глава 15, *Использование файловой системы*, знакомит нас с файловой системой движка Godot и показывает, как мы можем сохранять и загружать данные в нашей игре.

Глава 16, *Что дальше?*, знакомит нас с некоторыми последними приёмами и ресурсами, которые помогут начать следующий игровой проект, а также знакомит с сообществом разработчиков игр, частью которого мы можем стать.

Как извлечь максимальную пользу из этой книги:

Вам не нужны никакие предварительные знания о программировании или разработке игр. Единственное предварительное условие — вы открыты для обучения и готовы совершенствоваться. В книге я предлагаю несколько экспериментов, которые вы можете провести, и включил тесты для проверки ваших знаний. Важно, чтобы вы уделили время этому, чтобы знания закрепились в вашем мозгу.

Мы рассмотрим, как загрузить и настроить Godot Engine в первой главе этой книги, но вы уже можете загрузить Godot 4.2.1 или более позднюю версию, если вам не терпится. Все примеры в книге были протестированы на Godot 4.2.1, но должны работать и в будущих версиях.

Программное и аппаратное обеспечение, описанное в книге	**Требования к операционной системе**
Godot 4.2.1	Windows, macOS, or Linux
GDScript 2.0	

Godot Engine — очень лёгкое программное обеспечение, которое без проблем работает на старом, устаревшем оборудовании, но не помешает проверить минимальные технические требования и убедиться, что ваш компьютер им

соответствует: https://docs.godotengine.org/en/stable/about/system_requirements.html.

Если вы используете цифровую версию этой книги, мы советуем вам набрать код самостоятельно или получить доступ к коду из репозитория книги на GitHub (ссылка доступна в следующем разделе). Это поможет вам избежать возможных ошибок, связанных с копированием и вставкой кода.

Скачайте файлы примеров кода

Вы можете скачать файлы примеров кода для этой книги с GitHub по адресу https://github.com/PacktPublishing/Learning-GDScript-by-Developing-a-Game-with-Godot-4. Если в коде появятся обновления, они будут обновлены в репозитории GitHub.

У нас также есть другие пакеты кода из нашего богатого каталога книг и видео, доступных по адресу https://github.com/PacktPublishing/. Ознакомьтесь с ними!

Используемые условные обозначения

В этой книге используется ряд условных обозначений.

Код в тексте: указывает кодовые слова в тексте, имена таблиц базы данных, имена папок, имена файлов, расширения файлов, имена путей, фиктивные URL, пользовательский ввод и дескрипторы Twitter. Вот пример “В *Главе 1* мы научились писать код с помощью метода **_ready** узла.”

Блок кода задаётся следующим образом:

```
func deal_damage(amount: float) -> void:
   player_health -= amount
func heal(amount: float) -> void:
   player_health += amount
```

Когда мы хотим привлечь ваше внимание к определённой части блока кода, соответствующие строки или элементы выделяются жирным шрифтом:

```
func minimum(number1, number2):
   if number1 < number2:
      return number1
   else:
      return number2
```

Любой ввод или вывод командной строки записывается следующим образом:

```
unzip Godot_v4.2.1-stable_linux.x86_64.zip -d Godot
```

Жирный шрифт: обозначает новый термин, важное слово или слова, которые вы видите на экране. Например, слова в меню или диалоговых окнах отображаются **жирным шрифтом**. Вот пример: “Вы можете получить доступ к папке **user://** для данного проекта, открыв меню **Проект (Project)** и выбрав **Открыть папку пользовательских данных (Open User Data Folder)**.”

Контейнеры

Мы называем массив **контейнером (container)**, потому что мы можем хранить и извлекать из него фрагменты данных других типов, например, целые числа, строки, логические значения и т.д. Массив содержит другие данные.

Контейнеры структурируют другие данные, чтобы с ними было легче работать.

Свяжитесь с нами

Мы всегда рады отзывам наших читателей.

Общая обратная связь: Если у вас есть вопросы по любому аспекту этой книги, напишите нам по адресу customercare@packtpub.com и укажите название книги в теме сообщения.

Исправления (Errata)): Хотя мы приложили все усилия, чтобы обеспечить точность нашего контента, ошибки случаются. Если вы нашли ошибку в этой книге, мы будем признательны, если вы сообщите нам об этом. Пожалуйста, посетите www.packtpub.com/support/errata и заполните форму.

Пиратство: Если вы столкнётесь с нелегальными копиями наших работ в любой форме в Интернете, мы будем признательны, если вы предоставите нам адрес местонахождения или название веб-сайта. Пожалуйста, свяжитесь с нами по адресу copyright@packtpub.com со ссылкой на материал.

Если вы хотите стать автором: Если у вас есть тема, в которой вы разбираетесь, и вы хотели бы написать книгу или принять участие в её написании, посетите сайт authors.packtpub.com.

Поделитесь своими мыслями

После того, как вы прочтёте *Изучаем GDScript, разрабатывая игру с помощью Godot 4*, мы будем рады услышать ваши мысли! Пожалуйста нажмите здесь, чтобы перейти на страницу обзора Amazon для этой книги и поделитесь своим отзывом.

Ваш отзыв важен для нас и технического сообщества и поможет нам гарантировать предоставление контента высочайшего качества.

Загрузите бесплатную копию этой книги в формате PDF

Спасибо за покупку этой книги!

Вы любите читать на ходу, но не можете везде носить с собой печатные книги?

Приобретённая вами электронная книга несовместима с выбранным вами устройством?

Не волнуйтесь, теперь с каждой книгой Packt вы бесплатно получаете PDF-версию этой книги без DRM.

Читайте где угодно, в любом месте, на любом устройстве. Ищите, копируйте и вставляйте код из ваших любимых технических книг прямо в ваше приложение.

На этом преимущества не заканчиваются: вы можете получать эксклюзивный доступ к скидкам, информационным бюллетеням и отличному бесплатному контенту в своей электронной почте ежедневно.

Чтобы получить преимущества, выполните следующие простые шаги:

1. Отсканируйте QR-код или перейдите по ссылке ниже.



https://packt.link/free-ebook/978-1-80461-698-7

1. Предоставьте доказательство покупки
2. Вот и всё! Мы отправим вам бесплатный PDF и другие преимущества на вашу электронную почту.

Часть 1: Учимся программировать

В этой части мы начнём с загрузки бесплатного движка Godot Engine с открытым исходным кодом и настройки среды, в которой мы будем разрабатывать нашу собственную игру с нуля. Однако, прежде чем мы приступим к созданию игры, мы заложим прочные основы программирования с использованием языка программирования GDScript.

К концу этой части вы будете знать все о переменных, потоках управления, различных типах данных и контейнеров, методах и классах. Мы завершим эту часть главой о чистом коде.

Эта часть состоит из следующих глав:

- *Глава 1 , Настройка среды*
- *Глава 2 , Знакомство с переменными и потоком управления.*
- *Глава 3 , Группировка информации в массивах, циклах и словарях*
- *Глава 4 , Создание структуры с помощью методов и классов*
- *Глава 5 , Как и зачем поддерживать чистоту кода*

1

Настройка окружающей среды

Разработка игр становится более доступной, так как игровые движки становятся более мощными. Инструменты и конвейеры, которые были доступны только крупным компаниям и богатым людям, теперь доступны всем, у кого есть компьютер. Каждый может почувствовать удовлетворение от создания своей собственной игры и дать возможность другим поиграть в неё.

Это именно то, чего мы собираемся достичь в этой книге. Мы пройдём путь от полного отсутствия знаний о программировании или разработке игр до создания нашей самой первой игры и даже немного дальше.

В первой части этой книги мы узнаем всё о настройке Godot и программировании. Это может быть немного более абстрактно, но я постараюсь дать понятные примеры и удержать ваше внимание с помощью упражнений и экспериментов, которые вы можете провести самостоятельно.

Вторая часть этой книги будет гораздо более практичной, поскольку мы погрузимся с головой в создание нашей собственной видеоигры! Мы научимся использовать редактор Godot для создания интересных игровых сцен и сценариев.

В последней части этой книги мы выведем наши навыки программирования на новый уровень и узнаем всё о сложных темах, таких как более мощные концепции, шаблоны программирования, файловая система и многое другое.

А начнём мы это захватывающее приключение с создания нового проекта! Новый проект — это чистый лист с бесконечными возможностями. К концу этой главы мы

создадим свой собственный чистый лист и напишем наши первые строки кода. Но сначала я хотел бы уделить немного времени изучению игрового движка Godot и программного обеспечения с открытым исходным кодом в целом.

В этой главе мы рассмотрим следующие основные темы:

- Godot Engine и программное обеспечение с открытым исходным кодом
- Загрузка движка с официального сайта
- Создание нашего первого проекта
- Как присоединиться к сообществу

Технические требования

Поскольку эта книга направлена на то, чтобы вы могли перейти от нулевых знаний о программировании и разработке игр к среднему уровню, технических требований нет. Поэтому я просто проведу вас через все (или, по крайней мере, большинство) шагов, необходимых для создания игр.

Пример проекта и код

Пример проекта и код для этой книги можно найти в репозитории GitHub этой книги: https://github.com/PacktPublishing/Learning-GDScript-by-Developing-a-Game-with-Godot-4/tree/main/chapter01.

Игровой движок Godot и программное обеспечение с открытым исходным кодом

Мы будем использовать игровой движок Godot, о существовании которого вы, как я полагаю, уже знаете, поскольку эта книга посвящена именно этому движку. Но позвольте мне рассказать вам чуть больше о его истории и о

том, что означает open-source.

Немного информации о движке

Godot Engine это программное обеспечение с открытым исходным кодом, которое позволяет людям с любым опытом и из всех слоёв общества создавать игры. Проект был начат в 2007 году Хуаном Линецким и Ариэлем Манзуром как внутренний движок для нескольких аргентинских игровых студий. В конце 2014 года движок стал открытым, предоставив всем свободный доступ к коду. С тех пор он приобрёл большую популярность и в настоящее время является одним из самых используемых игровых движков на рынке. Многие коммерческие игры были выпущены или находятся в стадии разработки с использованием этого движка. Примерами выпущенных игр являются Brotato, Dome Keeper, Case of the Golden Idol и Cassette Beasts.

Для тех из вас, кто задаётся вопросом, да, движок назван в честь театральной пьесы *«В ожидании Годо»* Сэмюэля Беккета. Этот выбор названия обусловлен тем, что люди всегда будут ждать следующую версию или новую функцию, что приведет к бесконечному циклу ожидания.

Говоря о названии двигателя, давайте также разберёмся с произношением. Не существует стандартного способа произношения Godot. Из-за ассоциации с названием пьесы, которое написано на французском языке, некоторые люди говорят, что это должно быть «go-do», без ударения на какой-либо слог. Но большинство носителей английского языка сказали бы «GOH-doh» и сделали бы ударение на первый слог. Кроме того, есть некоторое количество людей, которые произносят его как «go-DOT», в основном потому, что это похоже на слово «robot», а логотип двигателя — синий робот. Но я заметил, что я произношу Godot каждый раз по-разному. Так что произносите его так, как вам нравится. Просто используйте примерно те же буквы.

Что такое программное обеспечение с

открытым исходным кодом?

Как упоминалось ранее, Godot — это проект с открытым исходным кодом, то есть исходный код движка находится в свободном доступе. Поскольку доступ есть у всех, люди могут изменять этот код по своему вкусу. После того, как они настроили достаточно параметров или разработали новую функцию, они могут попросить создателя программного обеспечения включить эти настройки или функции в исходный проект. Затем создатель проверит то, что сделал другой человек, немного изменит это, если необходимо, а затем добавит это в код исходного программного обеспечения. Этот процесс создаёт замкнутый круг, который приводит к ситуации, в которой все выигрывают:

- *Создатель программного обеспечения* может быстрее развивать код, поскольку все вносят свой вклад
- *Люди с техническими знаниями* могут добавлять недостающие им функции, подстраивая систему под свои нужды
- *Конечный пользователь* получает гораздо лучший и более стабильный конечный продукт.

Но не все проекты с открытым исходным кодом одинаковы. Каждое **бесплатное программное обеспечение с открытым исходным кодом (free open-source software) (FOSS)** поставляется с соответствующей лицензией. Эта лицензия диктует, как вы можете или должны использовать программное обеспечение. Некоторые из них довольно ограничительны, но в случае с Godot Engine нам повезло: мы можем делать все, что угодно, без существенных ограничений. Нам нужно только указать создателей в титрах наших игр.

Хорошо – мы знаем, что такое Godot Engine, как произносится его название (или нет), и почему FOSS такой классный. Давайте приступим к подготовке нашей среды разработки!

Получение и подготовка

Godot

Прежде чем мы сможем заняться программированием, нам нужно настроить среду разработки. Этим мы займёмся в этом разделе, начав с загрузки движка и создания нового проекта.

Загрузка движка

Приобрести двигатель относительно просто, для этого требуется выполнить всего несколько шагов:

1. Сначала нам нужно будет загрузить копию программного обеспечения. Это можно сделать по адресу https://godotengine.org/download.

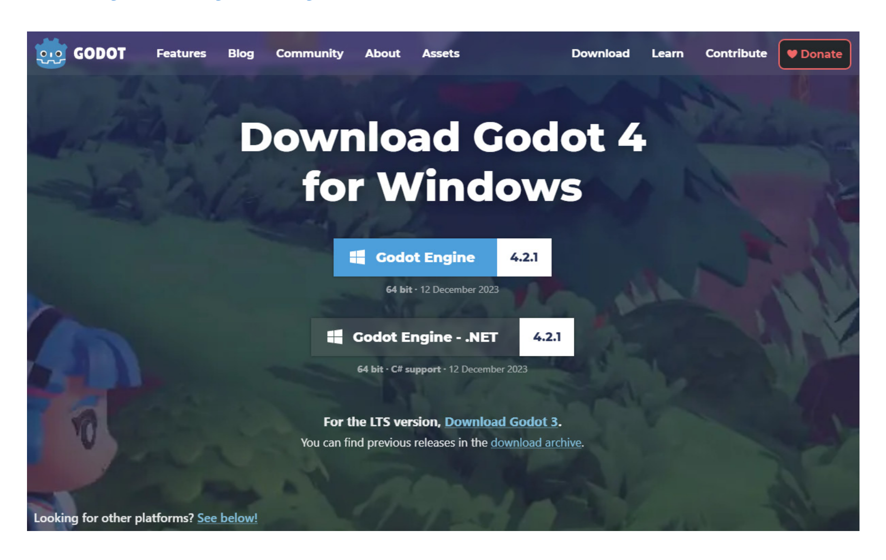


Рисунок 1.1 – Страница загрузки Godot Engine 4.0 для платформы Windows

1. Обычно страница автоматически направляет вас на страницу загрузки для той операционной системы, которую вы используете для просмотра веб-сайта, и вы можете нажать большую синюю кнопку в середине страницы, чтобы загрузить движок. Если этого не произошло, вам нужно будет выбрать платформу вашего

компьютера (Windows, macOS, Linux и т.д.) при прокрутке страницы вниз.

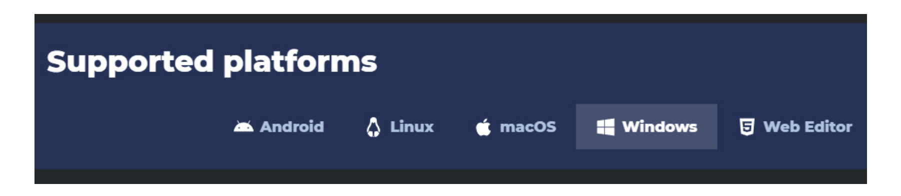


Рисунок 1.2 – Выберите платформу вашего компьютера, если страница загрузки не смогла её определить

1. Страница загрузки также должна определить, используете ли вы 64- или 32-битную систему. Если она не сделала этого правильно, то вы можете найти другие версии в разделе **Все загрузки (All downloads)**:

All downloads

Windows - Standard (x86_64)
64 bit
Windows - Standard (x86)
32 bit
Windows - .NET - Standard (x86_64)
64 bit · C# support
Windows - .NET - Standard (x86)
32 bit · C# support
Export templates
Used to export your games to all supported platforms
Export templates - .NET
Used to export your games to all supported platforms · C# support

Note: The 32-bit .NET binaries do not run on 64-bit Windows systems at the time being. Make sure to export 64-bit .NET binaries for your 64-bit target platforms.

Рисунок 1.3 – Раздел «Все загрузки», где вы можете найти различные версии движка

1. То, что мы скачали, — это ZIP-файл. Итак, распакуйте его, чтобы получить доступ к самому движку.

    - В Windows щёлкните правой кнопкой мыши по zip-файлу и выберите **«Извлечь все...» (Extract All...)**. Теперь следуйте всплывающим подсказкам, чтобы выбрать местоположение.
    - В macOS дважды щёлкните по zip-файлу, файл будет распакован в новую папку.
    - В Linux: выполните следующую команду в терминале:

    ```
    unzip Godot_v4.2.1-stable_linux.x86_64.zip -d G
    ```

5. Поместите извлеченные файлы куда-нибудь на вашем компьютере, где они будут в безопасности, например, на рабочий стол, в приложения или в любое другое место, кроме папки **«Загрузки» (Downloads)**. В противном случае, если вы похожи на меня, вы можете случайно удалить их во время чистки папки **«Загрузки»**.

Для этой книги мы будем использовать версию 4.0.0, так как она только что вышла. Но любая версия с 4 в начале должна работать нормально. К сожалению, это не гарантия. Мы сделаем всё возможное, чтобы поддерживать содержание этой книги в актуальном состоянии, но программное обеспечение с открытым исходным кодом может быстро обновляться.

Размер загрузки Godot Engine крошечный, около 30-100 МБ, в зависимости от вашей платформы. Этот небольшой пакет — всё, что нам нужно для создания потрясающих игр. Сравните это с 10 ГБ Unity и колоссальными 34 ГБ Unreal Engine! Конечно, всё это идет без каких-либо ресурсов, таких как визуальные эффекты или аудио.

Вот и все, что касается двигателя. Вам не нужно ничего устанавливать, чтобы его использовать.

Другие версии движка

Поскольку Godot Engine имеет открытый исходный код, существует также множество полных игровых проектов для него с открытым исходным кодом. Если вы когда-нибудь захотите запустить один из этих игровых проектов на своей машине, убедитесь, что вы используете правильную версию Godot; в противном случае игра может вылететь и могут произойти странные вещи. Вы можете найти и скачать все официальные версии Godot по адресу https://godotengine.org/download/.

Создание нового проекта

А теперь давайте продолжим и создадим наш первый проект Godot Engine, и, надеемся, в будущем появятся и другие!

1. Сначала откройте движок, дважды щелкнув файл, который мы скачивали в разделе *Загрузка движка*. Вас встретит такой экран:

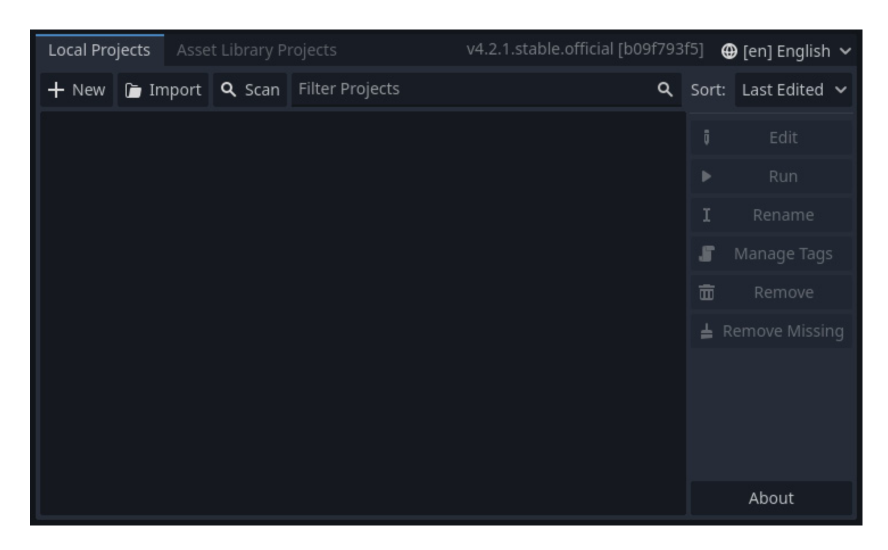


Рисунок 1.4 – Создание нового проекта нажатием кнопки «+ Новый»

1. Выберите + **Новый**; появится новое окно:

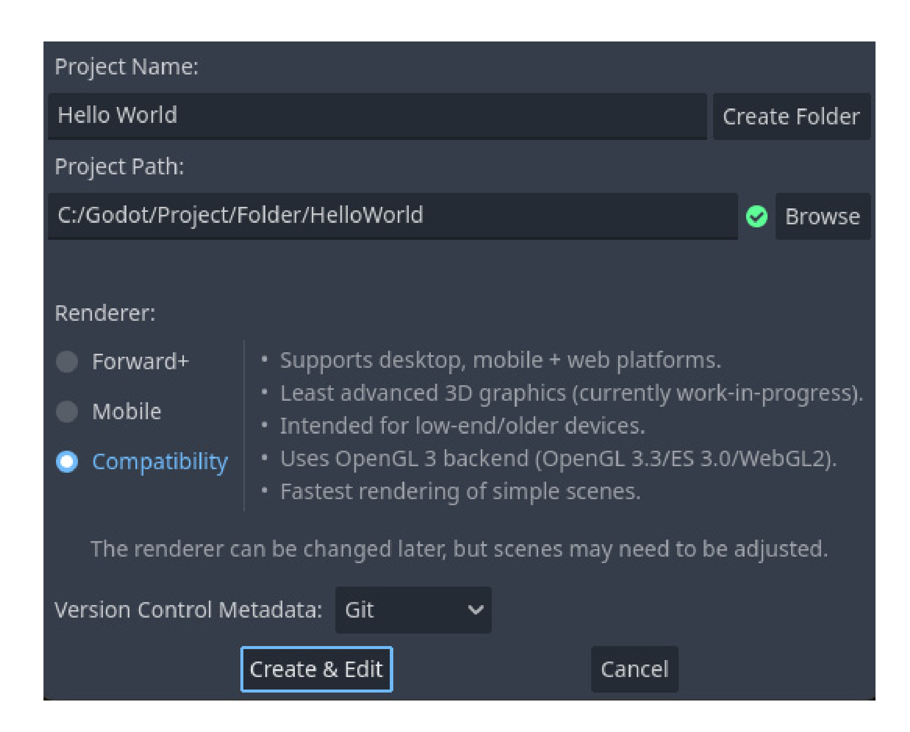


Рисунок 1.5 – Настройка нового проекта

1. Назовите проект **Hello World**.
2. Выберите область **Путь к проекту (Project Path)** , чтобы поместить проект. Создайте новую папку с помощью кнопки **Создать папку (Create Folder)** или используйте существующую, но учтите, что эта папка должна быть пустой. Хотя выбранная вами папка может уже содержать файлы, начав с чистого каталога мы сохраним всё, что делаем, в более организованном виде.
3. Выберите **Совместимость (Compatibility)** в категории **Отрисовщик (Renderer)**. Совместимый отрисовщик создан для того, чтобы наша игра могла работать на широком спектре оборудования и поддерживала старые видеокарты и веб-экспорт. Отрисовщик Forward + используется для передовой графики, но требует лучшей видеокарты, в то время как мобильный отрисовщик оптимизирован для мобильных устройств. Для того, что мы делаем, совместимый совместимый более чем достаточен и гарантирует, что мы можем экспортировать на максимально возможное количество платформ.

4. Наконец, нажмите **Создать и редактировать (Create & Edit)**!

Теперь Godot настроит базовую структуру нашего проекта в выбранной папке и через несколько секунд покажет нам редактор:

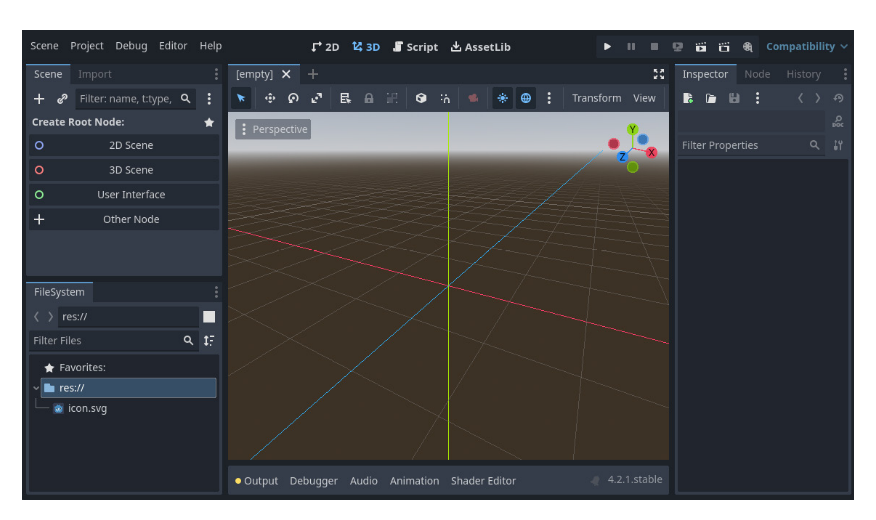


Рисунок 1.6 – Редактор Godot Engine 4.0

На первый взгляд это может показаться довольно пугающим — маленькие окна повсюду, множество элементов управления тут и там и гигантское трёхмерное пространство посередине. Не волнуйтесь. К концу этой книги вы будете знать все входы и выходы почти всего, что лежит перед вами. Вы в надёжных руках.

Забавный факт

Разработчики Godot использовали Godot Engine для создания самого редактора. Попробуйте вникнуть в это! Они сделали это, чтобы легко расширять и поддерживать редактор.

Светлая тема

Из-за ограничений печатных носителей тёмные скриншоты могут выглядеть зернистыми и нечеткими. Вот почему с этого

момента мы перейдем к светлой версии Godot. Нет никакой разницы, кроме внешнего вида редактора.

Если вы также хотите продолжить, используя светлую тему, выполните следующие необязательные действия:

1. Перейдите в **Редактор | Настройки редактора... (Editor | Editor Settings...)** в верхней части экрана:

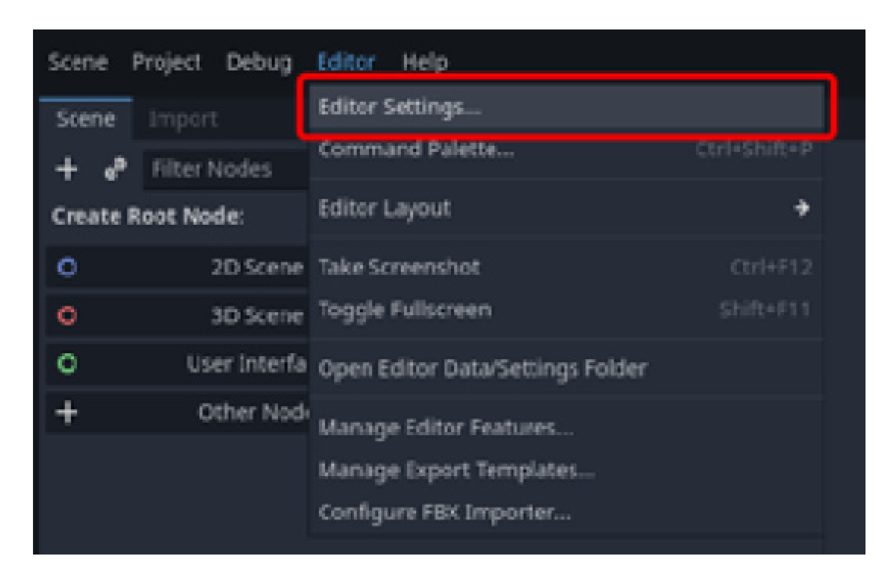


Рисунок 1.7 – Параметр «Настройки редактора...» в меню «Редактор»

1. Найдите пункт настроек **«Тема» (Theme)**.
2. Выберите тему **Light** в раскрывающемся списке **«Набор» (Preset)**:

Рисунок 1.8 – Выбор предустановленной темы «Светлая» (Light) в настройках темы

еперь редактор будет выглядеть так, как показано на *Рисунке 1.9*:

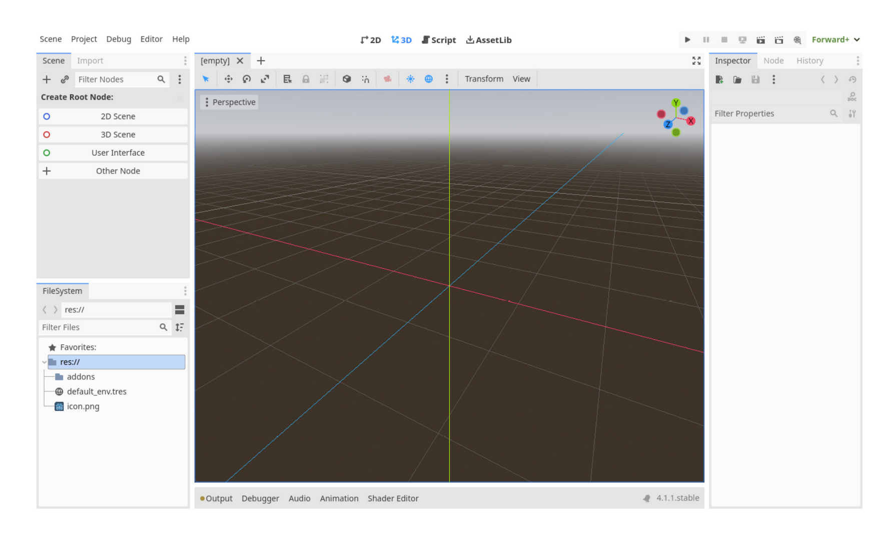


Рисунок 1.9 – Редактор Godot Engine с применённой темой «Светлая» (Light)

Разобравшись с этим, давайте вернёмся к созданию игры и научимся создавать сцену.

Создание главной сцены

Давайте продолжим и настроим нашу первую сцену:

1. На самой левой панели **Сцена (Scene)**, которая показана на *Рисунке 1.10*, выберите **2D сцена (2D Scene)**. Эта кнопка настроит сцену для 2D-игры, как показано здесь:

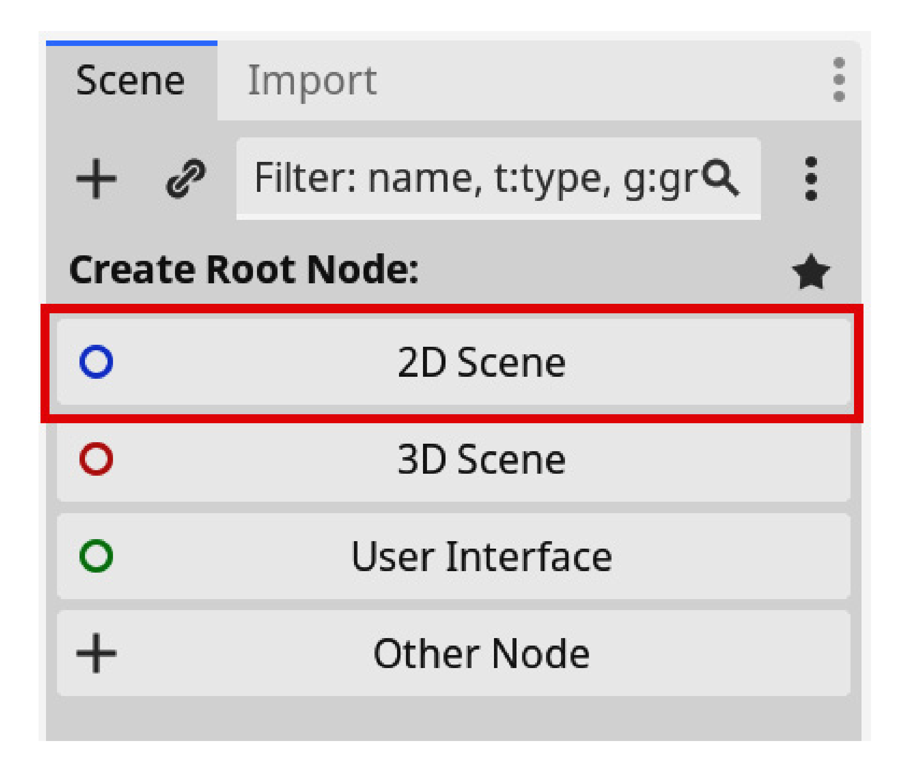


Рисунок 1.10 – Выбор опции «2D сцена» на левой панели

Вы увидите, что на панели **«Сцена» (Scene)** есть один узел под названием **Node2D**, а трёхмерное пространство в среднем окне заменено двухмерной плоскостью.

1. Щёлкните правой кнопкой мыши узел с именем **Node2D** и переименуйте его в **Main**. Этот узел будет нашим основным узлом для работы на данный момент:

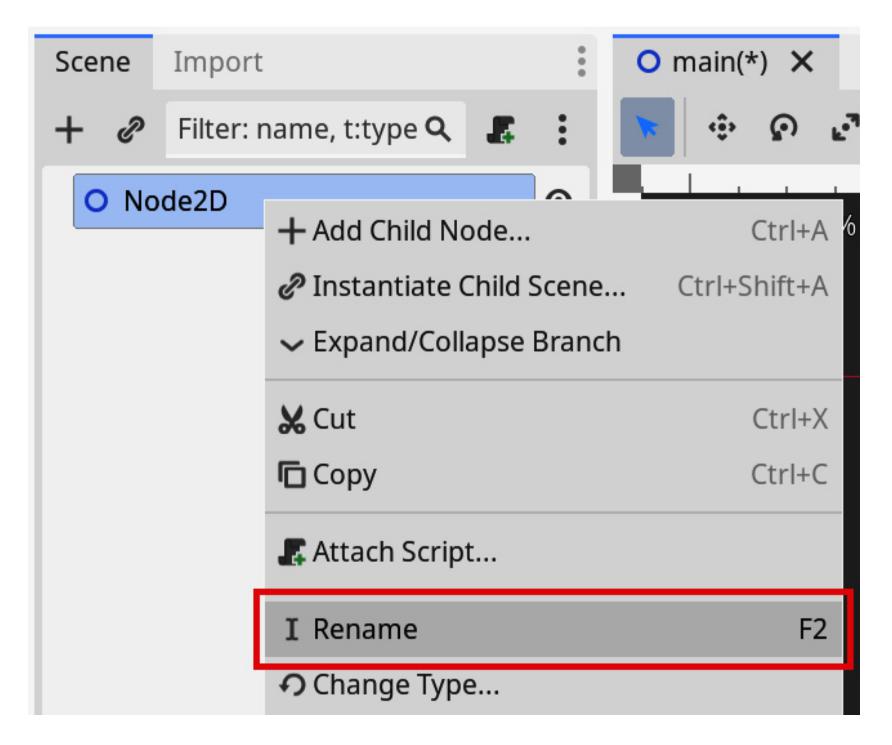


Рисунок 1.11 – Переименование узла Node2D в Main

1. Сохраните сцену, перейдя в **Сцена | Сохранить сцену (Scene | Save Scene)** или нажав *Ctrl/Cmd* + *S*:

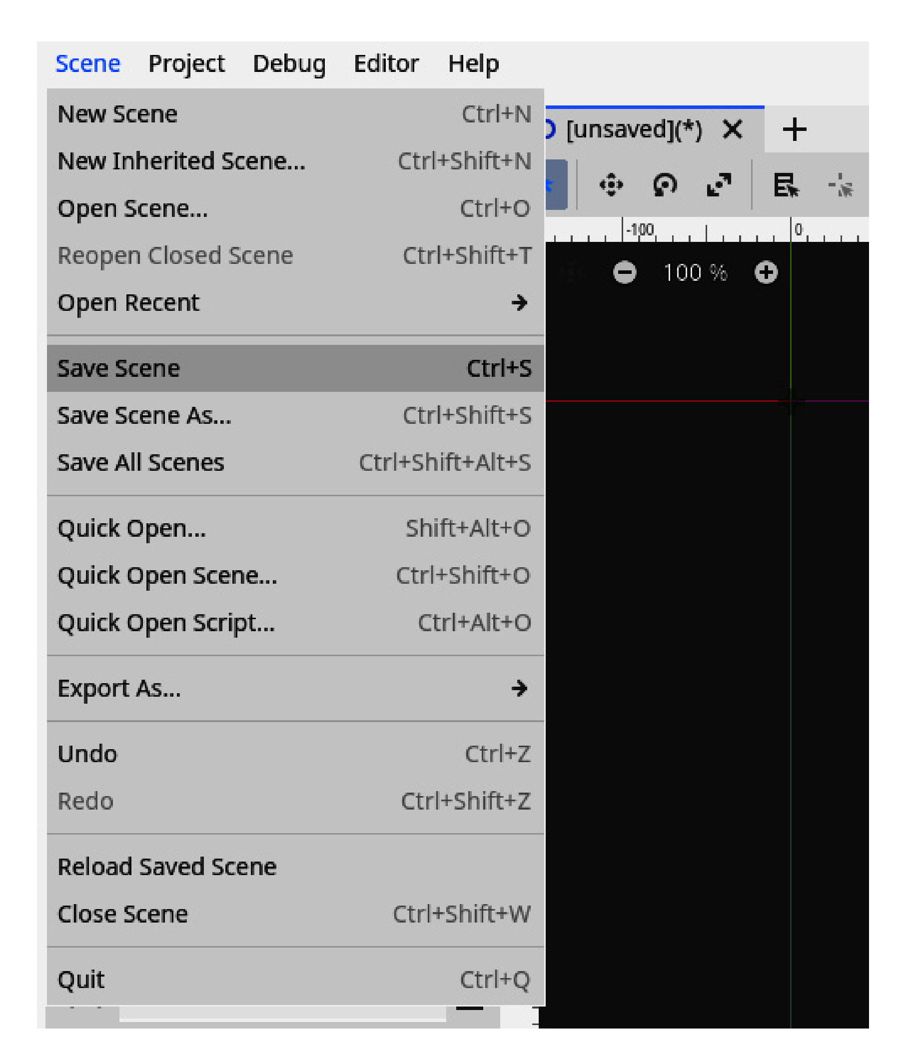


Рисунок 1.12 – Сохранение сцены

1. Нас спросят, где мы хотим сохранить сцену. Выберите корневую папку проекта и назовите файл **main.tscn**:

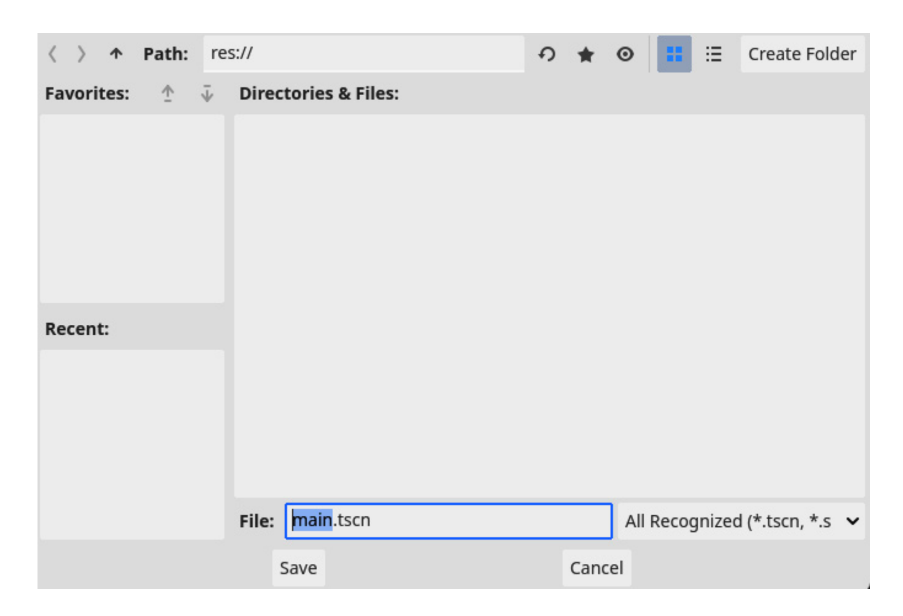


Рисунок 1.13 – Выбор корневой папки для сохранения сцены и присвоение ей имени main.tscn

Вот и всё, что касается создания нашей первой сцены. То, что мы только что добавили, — это узел. Эти узлы представляют всё в Godot. Изображения, звуки, меню, спецэффекты — всё является узлом. Вы можете думать о них как об игровых объектах, каждый из которых имеет отдельную функцию в игре. Игрок может быть узлом, так же, как враги или монеты.

С другой стороны, сцены — это наборы узлов или наборы игровых объектов. Пока что вы можете думать о сценах как об уровнях. Для уровня вам нужен узел игрока, несколько узлов врагов и куча узлов монет; набор этих узлов — это сцена. Как будто узлы — это краска, а сцены — наши холсты.

Мы ещё вернёмся к узлам и сценам на протяжении всей этой книги.

Краткий обзор пользовательского интерфейса (UI)

Сейчас самое время рассмотреть некоторые из наиболее

важных функций пользовательского интерфейса редактора. Как мы видели ранее, он выглядит примерно так:

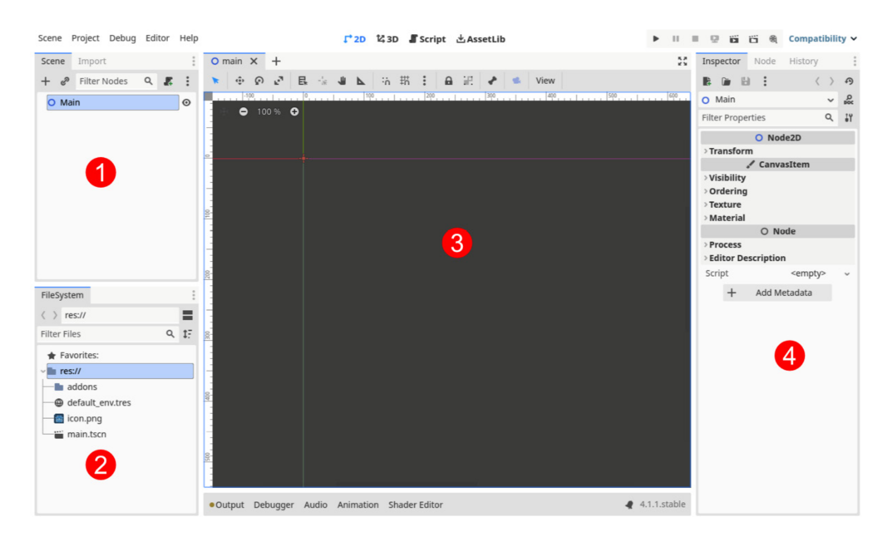


Рисунок 1.14 – Обзор редактора

Наиболее важными элементами редактора являются:

1. Область **«Дерево сцены» (Scene Tree)** показывает все узлы текущей сцены. На данный момент есть только один.
2. Область **«Файловая система» (FileSystem)** обеспечивает доступ к файлам в папке проекта.
3. Среднее окно — это **активный в данный момент главный редактор**. Сейчас мы видим 2D-редактор, который позволит нам размещать узлы в 2D-пространстве внутри сцены.
4. Область **«Инспектор» (Inspector)** находится у правого края и показывает свойства текущего выбранного узла. Если вы откроете некоторые меню-аккордеоны, например раздел **Transform**, вы увидите несколько настроек, связанных с выбранным узлом.

Узлы сами по себе не делают многого. Они предоставляют нам определенные функции, такие как показ изображения, воспроизведение звука и многое другое, но им нужна более высокоуровневая логика, чтобы связать их с реальной игрой.

Вот почему мы можем расширить их функциональность и поведение с помощью скриптов.

Пишем наш первый скрипт

Скрипт — это фрагмент кода, который добавляет логику к узлу, например, перемещает изображение или решает, когда воспроизводить тот или иной звук.

Сейчас мы создадим наш первый скрипт. Снова щелкните правой кнопкой мыши узел **Main** и выберите **«Прикрепить скрипт...» (Attach Script)**:

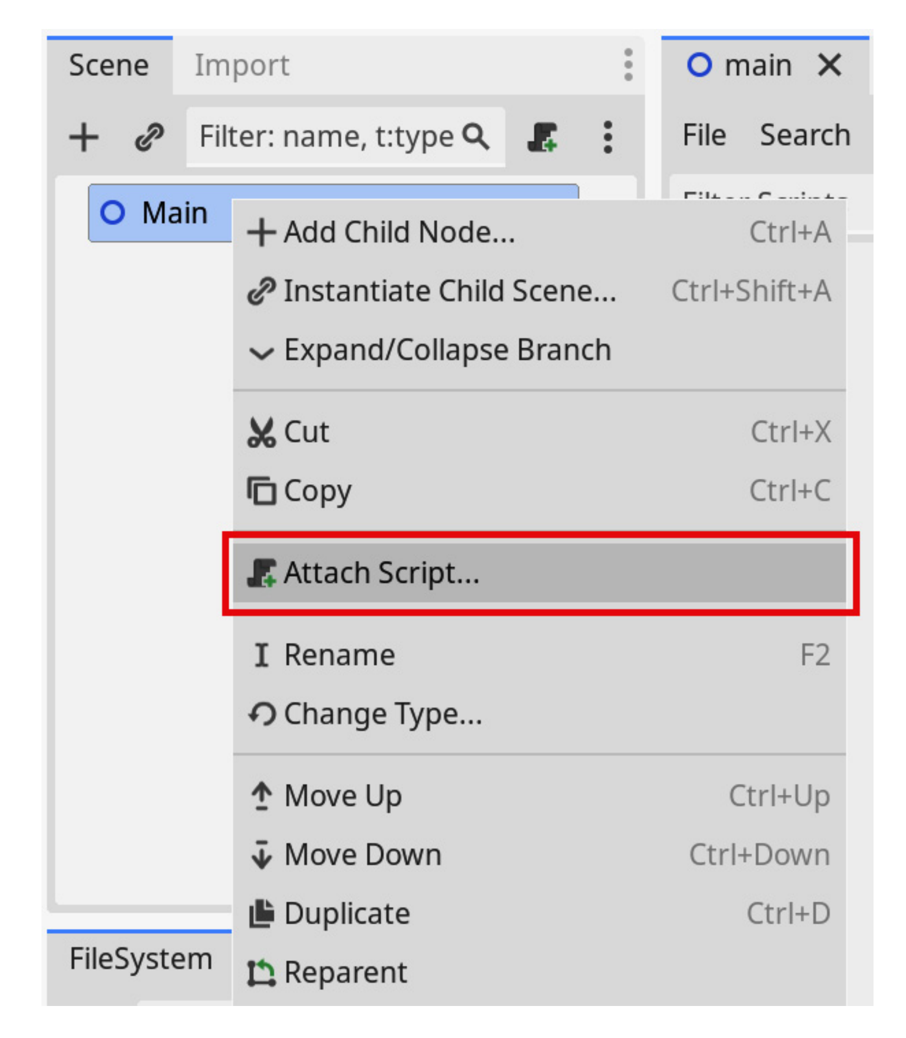


Рисунок 1.15 – Прикрепление скрипта к узлу Main

Появится всплывающее окно. Оставьте всё как есть. Важно отметить, что выбранный язык — **GDScript**, язык программирования, который мы изучим в этой книге. Остальное пока не так уж и важно. Имя скрипта оказалось предварительно созданным, на основании имени узла, к которому прикрепится этот скрипт. Нажмите **«Создать» (Create)**:

Рисунок 1.16 – Нажмите «Создать» для создания скрипта

Средняя панель, где раньше была 2D-плоскость, заменена новым окном:

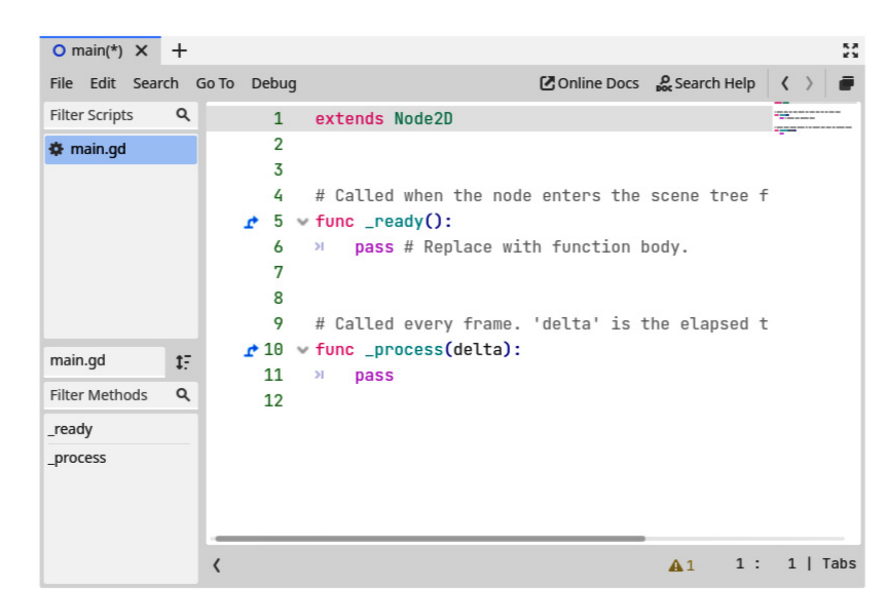


Рисунок 1.17 – Новый скрипт

Это редактор **скриптов**. На протяжении первой части книги мы проведем большую часть времени здесь, учась программировать.

Как вы могли заметить, среднее окно зависит от контекста. Это может быть **2D**, **3D** или редактор скриптов **Script**:

2D 3D Script AssetLib

Рисунок 1.18 – Различные главные окна

Для переключения между этими различными редакторами используйте кнопки в верхней части экрана.

Библиотека ресурсов AssetLib

Последняя вкладка, **AssetLib**, полезна для получения готовых ресурсов из библиотеки ресурсов Godot. Эта библиотека может предоставить пользовательские узлы, скрипты или любые другие ресурсы для вашего проекта непосредственно из редактора Godot. Мы не будем рассматривать 3D-редактор или AssetLib, но хорошо знать, что они там есть.

Все ресурсы библиотеки ресурсов **AssetLib** имеют открытый исходный код и, таким образом, полностью бесплатны для использования! Ура FOSS!

Если вы пробовали менять редакторы, вернитесь в редактор скриптов **Script**, чтобы мы могли создать наш первый скрипт и убедиться, что всё готово. Код внутри скрипта на данный момент выглядит так:

```
extends Node2D
# Вызывается, когда узел впервые попадает в дерево сцены
func _ready():
   pass # Заменить телом функции.
# Вызывается каждый кадр. 'delta' — это время, прошедшее
```

```
func _process(delta):
   pass
```

Опять же, не беспокойтесь обо всех этих различных командах и специфическом синтаксисе. Мы рассмотрим всё в свое время. На данный момент достаточно знать, что это скрипт, написанный на GDScript — скриптовом языке Godot.

Чтобы создать классическую программу “**Hello, World**”, которая является основной для начинающих программистов, все, что нам нужно сделать, это изменить строку, содержащую **pass # Заменить телом функции.** на следующую:

```
   print("Hello, World")
```

Эта строка кода покажет текст "**Hello, World**", она не будет использовать принтер для печати чего-либо. Мы также можем выбросить часть кода, который нам не нужен. Весь скрипт теперь должен выглядеть так:

```
extends Node2D
func _ready():
   print("Hello, World")
```

Обратите внимание, что перед оператором печати **print**, который мы добавили, должна быть *табуляция (tab)*. Мы добавляем эту *табуляцию*, потому что она показывает, что строка кода принадлежит функции **_ready**. Мы называем практику добавления *табуляции* in перед строками **отступом (indentation)**.

Важная заметка

В тексте этой книги мы не использовали табуляцию по редакционным причинам. Мы будем использовать три пробела для обозначения одной табуляции. Вот почему вам лучше не копировать и не вставлять код из этой книги в редактор. Полный код для этой книги можно получить и скопировать из репозитория GitHub этой книги (ссылка в разделе *Технические*

требования).

Все строки в функции **_ready** будут выполнены, когда узел будет готов, мы увидим, что это значит более подробно позже. На данный момент достаточно знать, что эта функция выполняется, когда узел готов к использованию.

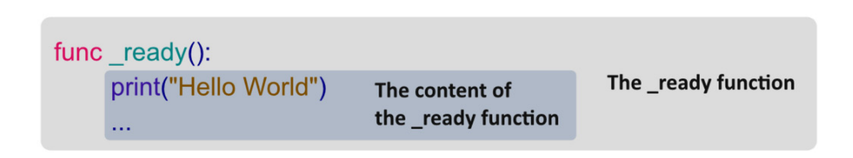


Рисунок 1.19 – Функция содержит блок кода

Функции — это небольшие группы кода, которые может выполнить компьютер. Функция всегда вводится ключевым словом **func**, за которым следует имя функции.

Вы можете видеть, что предварительно заполненный скрипт также предоставил нам функцию **_process**, которую мы не будем использовать сейчас, поэтому мы удалили ее. Мы вернемся к функциям в *Chapter 4*. Помните, что каждая строка кода в функции **_ready** будет выполняться с момента запуска нашей игры, и что этим строкам должна предшествовать *табуляция*.

Используйте клавишу *Tab* для вставки этих вкладок. Символ на вашей клавиатуре выглядит так: ⭾

Последняя интересующая нас строка в скрипте говорит **extends Node2D**. Это означает, что мы используем **Node2D**, тип узла, который мы добавили к сцене, как основу для запуска скрипта. Всё в скрипте является расширением функциональности, которую предоставляет **Node2D**. Мы узнаем больше о расширении скриптов и классов в *Chapter 4*.

Теперь нажмите кнопку запуска в правом верхнем углу, чтобы запустить наш проект:



Рисунок 1.20 – Кнопка запуска используется для запуска проекта

Всплывающее окно спросит нас, какую сцену мы хотим использовать в качестве основной. Выберите **«Выбрать текущий» (Select Current)**, чтобы установить текущую сцену в качестве основной:

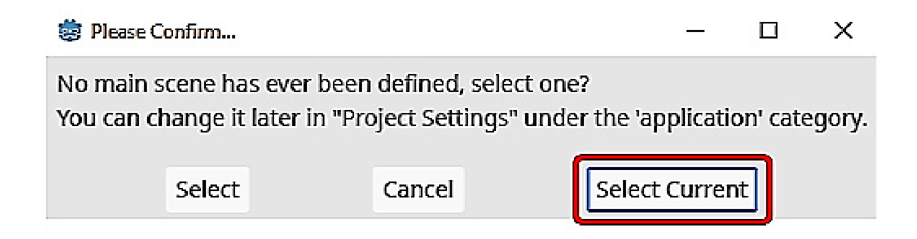


Рисунок 1.21 – Godot Editor попросит нас определить главную сцену. Мы можем просто выбрать текущую

Появится пустой серый экран. Мы пока ничего визуально не добавили в нашу игру. Позже здесь будет размашистая и захватывающая игра. Но этот серый экран — то, чего нам следует ожидать сейчас:



Рисунок 1.22 – Пустое окно игры

Самая захватывающая часть сейчас происходит в самом окне редактора. Вы увидите новое маленькое окно, разворачивающееся снизу, где печатается текст **Hello, World**:



Рисунок 1.23 – Вывод игры: Hello, World

Успех! Мы написали наш первый скрипт!

В качестве эксперимента попробуйте изменить текст в двойных кавычках *шага 4* и перезапустите программу. Вы должны увидеть новый текст, напечатанный в окне вывода:



Рисунок 1.24 – Вывод игры после изменения печатного текста

Это были наши первые шаги в создании сцены и сценария в игровом движке Godot. В этой книге мы узнаем все, что нам нужно знать, чтобы создать целую игру с нуля, но пока оставим это здесь. Далее мы кратко рассмотрим присоединение к сообществу разработки игр.

Документация по движку Godot

Если вы когда-нибудь запутаетесь, есть также официальная **документация по Godot Engine**. Это очень исчерпывающий источник информации обо всех различных классах и узлах. Также он содержит руководства по всем различным подсистемам, связанным с движком.

Доступ к документации можно получить здесь: https://docs.godotengine.org/.

Всякий раз, когда вы ищете, как использовать определённую часть движка, или что-то в книге вам не совсем понятно, вы можете обратиться к документации.

Присоединяйтесь к нашему сообществу!

В качестве последней части этой главы я приглашаю вас присоединиться к нашему сообществу! Если вам нужна помощь, вы столкнулись с ошибкой или просто хотите пообщаться с другими разработчиками игр, приходите и найдите нас на любой из платформ, упомянутых на странице https://godotengine.org/community.

Я также призываю вас публиковать свои успехи на 𝕏, Facebook, Instagram, Mastodon или любой другой платформе социальных сетей. Получать обратную связь и дополнительные взгляды на ваши проекты всегда весело! Если вы решите это сделать, не забудьте использовать эти хэштеги: **#GodotEngine**, **#indiedev** и **#gamedev**.

Хотите связаться со мной лично? Посетите мой сайт для получения самой актуальной контактной информации: www.sandervanhove.com.

В последней части этой книги я расскажу подробнее о сообществе и о том, как вы можете присоединиться и, возможно, даже помочь. Но сейчас давайте сосредоточимся на изучении ремесла самостоятельно!

Итоги

В этой главе мы узнали о Godot Engine, который является FOSS. Затем мы скачали движок для себя и создали наш первый проект. Наконец, мы увидели, что встроенный язык программирования — GDScript, и создали наш первый скрипт "**Hello, World**".

В следующей главе мы начнем наше путешествие по изучению программирования. Увидимся там!

Опрос

- Что означает аббревиатура FOSS и где она используется?
- Является ли движок Godot проектом с открытым

исходным кодом?
- Какую строку кода мы добавили, чтобы показать *“Hello, World”* в окне «Вывод» (Output)? Зачем мы добавили *табуляцию* в начале этой строки?
- Что такое *узлы (nodes)* в движке Godot и как они связаны со *сценами (scenes)*?

2

Знакомство с переменными и потоком управления

Программирование — это работа с данными. В играх эти данные могут быть положением игрока, количеством его здоровья или предметами, которыми он владеет. Эти фрагменты данных называются **переменными (variables)**, и в этой главе мы узнаем, как работают эти переменные и как мы можем ими управлять.

Во второй части этой главы мы узнаем, как использовать эти переменные для принятия решений во время игры. Например, если здоровье игрока падает до нуля, мы заканчиваем игру. Эта концепция называется **потоком управления (control flow)**, потому что мы контролируем поток на протяжении всего кода.

В этой главе мы рассмотрим следующие основные темы:

- Что такое переменные?
- Типы данных – целые числа (Integers), числа с плавающей запятой (floats) и строки (strings)
- Что такое константы?
- Начало работы с потоком управления
- Комментирование в коде

Технические требования

Мы подготовили всё необходимое для *Главы 2*. Если вы где-то застрянете, проверьте папку **chapter02** в репозитории примеров кода. Репозиторий можно найти здесь: https://github.com/PacktPublishing/Learning-GDScript-by-Developing-a-Game-with-Godot-4/tree/main/chapter02.

Что такое переменные?

По секрету, под капотом, игры — это данные. Положение персонажа игрока, имена ваших товарищей по команде в онлайн-матче, уровень громкости в окне настроек или максимальное расстояние, которое может преодолеть огненный шар — всё это фрагменты данных, с которыми работает игра, чтобы создать игровой опыт, который нравится пользователям.

Например, положение персонажа игрока может меняться во время игры, в зависимости от нажатий клавиш со стрелками на клавиатуре. Компьютер примет этот ввод, вычислит новое положение и покажет игроку, где находится его персонаж в виртуальном мире. Компьютер будет делать это в той или иной степени для каждого фрагмента данных в игре. Мы узнаем гораздо больше о том, как компьютер выполняет все эти вычисления на протяжении всей книги, но сначала нам придется узнать об этих маленьких фрагментах данных, где они хранятся на компьютере и как они используются во время игры.

В коде игры каждый из этих фрагментов данных представлен переменной. Давайте подробнее рассмотрим переменные в следующем разделе.

Переменные – ящики в картотеке, полные данных

Отличный способ представить переменные — рассматривать их как ящики в картотеке. Каждый ящик представляет собой одну переменную и содержит данные для этой переменной. Если мы хотим изменить некоторые из этих данных, мы открываем правый ящик, меняем значение и кладём результат обратно в ящик. Конечно, нам нужно знать, какой ящик содержит данные о позиции игрока, а какой — количество оставшихся у игрока жизней. Смешивание этих данных привело бы к очень запутанным ситуациям. Умные архивисты решили эту проблему, поместив бирки с именами на каждый ящик, чтобы

они могли одним взглядом увидеть, какие данные содержит каждый из них.



Рисунок 2.1 – Представление переменных в виде ящиков шкафа

Тот же принцип применим к переменным. При создании переменной мы начинаем с того, что даем ей имя. Процесс создания резервирует *ящик*, немного места в памяти компьютера, к которому мы можем обратиться, используя данное нам имя. Затем мы можем использовать это имя для извлечения значения этой переменной из памяти и сохранения новых значений в том же ящике.

Именование переменных

Мы уже говорили о важности имени переменной для резервирования места и извлечения данных переменной. Они должны быть названы таким образом, чтобы мы могли легко их использовать. Однако есть некоторые дополнительные правила, которым должны соответствовать имена переменных, чтобы они были допустимыми.

Использование правильных символов

Имена переменных в GDScript не могут содержать пробелы. Они могут состоять только из букв, цифр или символов подчёркивания. Другие символы приведут к ошибкам. Поэтому безопасными символами для использования являются **abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234**

Обратите внимание, что имя не может начинаться с цифры. Вы также не можете выбрать ни одно из встроенных ключевых слов языка программирования в качестве имени переменной, поскольку это будет чрезвычайно запутанным и для вас, и для интерпретатора.

Важные заметки

Ключевые слова: Некоторые слова имеют особое значение в GDScript. Они выполняют определённые команды – например, создание переменной. Мы называем эти слова **ключевыми словами**. Мы увидим их гораздо больше в книге. Имена переменных не могут быть существующими ключевыми словами в GDScript.

Интерпретатор: Часть движка, которая запускает код, называется **интерпретатором**. Он проходит по коду GDScript, строка за строкой, и интерпретирует его как машиночитаемый код. Мы поговорим об этом подробнее в *Главе 4*.

Соглашение об именовании переменных в GDScript заключается в том, чтобы не использовать заглавные буквы и использовать подчёркивания для разделения слов; этот способ именования переменных называется **snake case** или **snake_case** — змеиная нотация. Я знаю, знаю — я только что сказал вам, что заглавные буквы являются допустимыми символами для использования в именах переменных. Но соглашение GDScript заключается в том, чтобы (почти) никогда не использовать их, потому что они используются в других случаях, например, в именах констант и классов. Я объясню различные соглашения об именовании и кодировании далее в *Главе 5*. На данный момент полезно знать, что соглашение о кодировании — это руководство по тому, как стилизовать ваш код. Оно ничего не

говорит о логике кода, а только о том, как он написан.

Использование описательных имён

Умные программисты дают каждой переменной описательное имя, чтобы мы могли сразу увидеть, какие данные она содержит. Используйте имя переменной, например, **number_of_lives**, а не что-то вроде **nol**, **l** или **lives**. Хотя при создании переменной вы будете знать, какие данные она содержит, вы забудете об этом, когда вернетесь к тому же коду через месяц или даже неделю. Не беспокойтесь о длине имени переменной. Знание того, какие данные она содержит и как они используются, имеет большее значение.

Короче говоря, хорошие имена переменных должны выглядеть так:

```
var player_position # Позиция игрока
var car_acceleration # Ускорение машины
var distance_to_goal # Расстояние до цели
```

Они не должны выглядеть так:

```
var a
var thing
var PLaYerpOsition
var distancetogoal
```

Теперь, когда мы знаем, как называть переменные, давайте в следующем разделе рассмотрим, как можно создавать и назначать переменные в GDScript..

Переменные в GDScript

Давайте подробнее рассмотрим переменную в скрипте. Создание и назначение переменной в GDScript выглядит так:

```
var number_of_lives = 3
```

В этом примере есть четыре важных шага:

1. Сначала мы указываем, что создаём новую переменную, используя ключевое слово **var**.
2. Во-вторых, мы указываем имя, которое мы хотим присвоить этой переменной — в нашем случае, **number_of_lives**. Из имени переменной мы знаем, что данные, содержащиеся в переменной, должны быть количеством жизней.
3. В-третьих, мы используем знак равенства, чтобы указать, что мы хотим присвоить значение нашей только что созданной переменной.
4. Наконец, после знака равенства мы указываем, какие данные следует поместить в переменную — в этом примере это число **3**.

Стоит отметить, что присваивания работают справа налево. Это означает, что значение справа (в данном случае **3**) будет присвоено переменной слева (**number_of_lives**).

Важная заметка

Обратите внимание, что назначение переменной — это не то же самое, что математическое уравнение. Математическое уравнение, например, **x = 3**, означает, что обе стороны равны. В то время как назначение переменной, например **number_of_lives = 3**, на самом деле сохраняет значение с правой стороны в переменной слева.

В следующем разделе мы увидим,как можно распечатать переменные.

Печать переменных

До сих пор мое объяснение переменных было абстрактным. Итак, давайте потратим немного времени на создание первого скрипта, который использует переменные!

Откройте скрипт **main.gd**, который мы создали в *Главе 1* и

добавьте строку, которая создаёт переменную для количества жизней после строки, которую мы использовали для вывода *Hello, World* в *строке 7*, например:

```
var number_of_lives = 3
```

Опять же, убедитесь, что вы добавили символ табуляции в начало строки. Обычно редактор добавляет его автоматически.

Если мы запустим эту программу, вы не увидите большой разницы с того момента, когда мы запускали её в *Главе 1*. Это потому, что создание переменных ничего не меняет в выполнении игры, пока мы что-то с ними не сделаем.

Я слышу, как вы думаете: «Но разве мы не можем использовать команду **print()**, которую мы использовали ранее, чтобы вывести значение переменной?» Можем, так что давайте сделаем это:

```
print(number_of_lives)
```

Теперь полный код выглядит так:

```
3
4 func _ready():
5 	print("Hello World")
6 	
7 	var number_of_lives = 3
8 	print(number_of_lives)
9
```

Рисунок 2.2 – Печать переменной number_of_lives можно легко выполнить с помощью функции print

При запуске программы мы видим следующее:

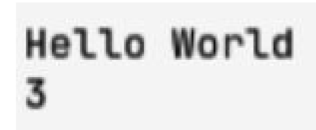


Рисунок 2.3 – Результат после печати переменной number_of_lives

Этот результат уже отличный, но мы можем сделать больше! Мы можем передать несколько фрагментов данных в одну и ту же команду **print()**и она выведет их все на одной строке в том порядке, в котором мы их указали. Попробуйте использовать эту строку в нашем скрипте:

```
print("Количество жизней равно: ", number_of_lives)
```

Убедитесь, что вы ставите запятую между каждым фрагментом данных при предоставлении их команде **print()**. Мы называем фрагмент данных, который мы предоставляем функции, **аргументом**.

Команда **printt()** выведет все значения, как и обычная команда **print()**, но вставит символы табуляции между значениями. Эта команда — один из моих методов отладки, но об этом подробнее в *Главе 3*:

```
printt("У игрока осталось", number_of_lives, "жизней из"
```

В качестве эксперимента попробуйте напечатать что-нибудь с помощью команды **prints()**. Вы можете использовать её так же, как и обычные команды **print()** и **printt()**. Однако чем они отличаются?

Если вы провели эксперимент, то увидели, что функция **prints()** работает так же, как **printt()**, однако она вставляет пробелы между переданными аргументами.

Теперь, когда мы знаем, как распечатать переменные для

наблюдения за их значениями, давайте посмотрим, как изменить значения, которые содержат эти переменные.

Изменение значения переменной

В отличие от данных в картотечных шкафах, данные в видеоигре, как правило, обновляются часто, очень часто, а иногда и сотни раз в секунду. К счастью, синтаксис обновления переменной ещё проще, чем её создание:

```
var number_of_lives = 3
number_of_lives = 4 + 1
```

Предыдущий код создаёт переменную **number_of_lives** и присваивает ей значение **3**, как и раньше. Далее, шаг за шагом, происходит следующее:

1. Сначала мы указываем, какую переменную мы используем, чтобы компьютер знал, какой ящик картотечного шкафа открыть.
2. Далее мы используем оператор присваивания, знак равенства, который сигнализирует о том, что мы хотим изменить значение этой переменной на новое.
3. Наконец, мы указываем новое значение — в данном случае результат **4 + 1**. Компьютер выполнит для нас вычисления, сделает вывод, что результат равен **5** и присвоит это значение переменной.

Подводя итог этим двум строкам кода, мы создали переменную с именем **number_of_lives**, присвоили ей значение **3**, а в следующей строке присвоили этой же переменной значение **5**.

Вы можете продолжить, используя другие переменные или даже ту же самую переменную для присваивания значений, например:

```
var number_of_lives = 3 # количество жизней
var fireball_damage = 2 # урон от огненного шара
```

```
number_of_lives = number_of_lives - fireball_damage
```

В конце этого кода переменная **number_of_lives** будет содержать значение **1**. Вот что происходит после знака равенства в третьей строке:

Компьютер видит два имени переменных, ищет их и заменяет их значениями. Он заменит **number_of_lives - fire_ball_damage** на **3 - 2**.

Как и в предыдущем примере, компьютер выполняет вычисления, приходит к выводу, что результат равен **1** и присваивает это значение переменной **number_of_lives**.

Расширьте наш скрипт предыдущим примером и снова выведите **number_of_lives**, чтобы увидеть результат:

```
func _ready():
	print("Hello World")

	var number_of_lives = 3
	print(number_of_lives)

	var fire_ball_damage = 2
	number_of_lives = number_of_lives - fire_ball_damage
	print("The number of lives is: ", number_of_lives)
```

Рисунок 2.4 – Расчёт нового количества жизней, которыми обладает игрок при попадании в него огненного шара

Хорошо, теперь мы можем изменить значение переменной. В следующем разделе мы рассмотрим, какие математические операции нам доступны.

Математические операторы

В предыдущих примерах мы видели некоторые математические операторы, добавляющие и вычитающие числа с помощью операторов + и **-**. Мы называем символы, такие, как + и **–**

операторами, потому что они берут данные, например числа, и работают с ними. GDScript предоставляет нам все ожидаемые математические операторы, которые работают примерно так же и в том же порядке, что и в обычной математике.

Operator	**Example**	**Result**
Возведение в степень (Power)	**5 ** 3**	**125**
Умножение (Multiplication)	**8 * 2**	**16**
Деление (Division)	**16 / 8**	**2**
Целочисленный остаток от деления по модулю (Modulo)	**8 % 3**	**2**
Сложение (Addition)	**2 + 4**	**6**
Вычитание (Subtraction)	**6 - 2**	**4**

Таблица 2.1 – Математические операторы в порядке их применения при выполнении

Как и в обычной математике, мы можем использовать скобки, чтобы сначала выполнить часть уравнений, например: **(4 + 3) * 7**.

Узнайте больше

Хотите узнать больше об операторах в GDScript? Ознакомьтесь с официальной документацией: https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_basics.html#operators

В качестве эксперимента попробуйте разделить число на ноль и посмотрите, что получится. Если вы не можете закрыть окно игры, пока работает программа, вы можете нажать кнопку **«Остановить запущенный проект» (Stop Running Project)** (см. *Рисунок 2.5*), чтобы принудительно остановить её.

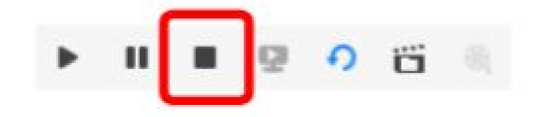


Рисунок 2.5 – Кнопка «Остановить запущенный проект»

Математические операторы являются основой всех манипуляций данными в программировании. Есть способ записать эти математические уравнения более компактно. Мы

рассмотрим один из таких методов далее.

Другие операторы присваивания

Мы присвоили новые значения переменным с помощью оператора присваивания (=), но есть и другие вариации этого оператора. Например, вы хотите прибавить число к переменной и сохранить результат в этой же переменной — например, отслеживая количество жизней после того, как игрок исцеляется от урона:

```
number_of_lives = number_of_lives + 2
```

Это настолько часто встречающийся вариант использования, что для него есть специальные операторы – +=, -=, *=, /=, **%**= и **=. Если мы перепишем предыдущий пример с одним из этих операторов, то получим следующее:

```
number_of_lives += 2
```

В этом коде говорится: возьмите значение **number_of_lives**, добавьте к нему число **2** и присвойте результат той-же переменной **number_of_lives**.

Другие операторы присваивания следуют той же структуре, только с другими операциями, такими как вычитание, умножение, деление, модуль и возведение в степень. Обратите внимание, что вы не можете использовать эти специальные операторы присваивания при создании переменной, поскольку она ещё не будет существовать.

В качестве эксперимента попробуйте использовать одно из этих специальных назначений при создании переменной и посмотрите, какая ошибка возникнет.

Если мы используем оператор -= для вычитания урона от огненного шара из количества жизней, то получим такой код:

```
func _ready():
	print("Hello World")

	var number_of_lives = 3
	print(number_of_lives)

	var fire_ball_damage = 2
	number_of_lives -= number_of_lives
	print("The number of lives is: ", number_of_lives)
```

Рисунок 2.6 – Использование оператора присваивания -= для вычитания урона от огненного шара из здоровья игрока

Мы узнали всё о переменных и о том, как их изменять во время выполнения нашей игры. В следующем разделе мы рассмотрим различные типы данных, которые мы можем хранить в этих переменных.

Типы данных – целые числа (Integers), числа с плавающей запятой (floats) и строки (strings)

До сих пор мы работали только с числами, но переменные могут содержать множество различных типов данных. В этом разделе мы рассмотрим три наиболее используемых типа данных. На протяжении всей книги мы увидим ещё больше типов, когда они будут применяться.

Целые числа (Integers)

Целые числа — это целые числа, например **1**, **-422** или **10983457**. Они не содержат десятичной точки. Они очень хорошо подходят для подсчета количества жизней или пуль.

Целое число в GDScript может быть любым значением от **-9223372036854775808** до **9223372036854775807**. Это ограничение накладывается способом представления чисел в компьютере с использованием битов; это не произвольное число, налагаемое самим Godot. Число будет циклически изменяться и переключаться с очень, очень высокого на очень, очень низкий, если вы пересечете этот барьер в любом направлении, что не рекомендуется. К счастью, этот диапазон дает нам больше чисел, чем количество звезд в нашей галактике на несколько величин, так что мы должны быть в безопасности:

```
var big_number = 9223372036854775807
big_number = big_number + 1
```

После выполнения этого кода наша переменная, **big_number** будет содержать наименьшее возможное число в целочисленном типе GDScript!

Числа с плавающей запятой (Floats)

Числа с плавающей запятой, также известные как **float**, это числа, это числа, содержащие десятичную точку (также известные как **числа с плавающей точкой**), например **3.41**, **-1978.8791** или **1.0**. Они очень хорошо подходят для обработки более конкретных чисел, скоростей, положений или оценок действительных чисел, например, числа Пи. В GDScript float имеет точность до **14** десятичных цифр.

Важная заметка

Целые числа можно хранить в float, если они достаточно малы, без проблем. Однако, когда мы сохраняем float в integer, всё после десятичной точки будет отброшено! Это называется **неявным преобразованием типа (implicit type conversion)**, потому что мы неявно преобразуем float в integer.

Любопытно знать, что компьютеры очень плохо справляются с числами с плавающей запятой. Из-за особенностей работы

компьютеров они не могут точно представить все возможные числа и, таким образом, в какой-то момент им придется округлять их. Это приводит к незначительным отклонениям от того, что вы ожидаете получить в результате вычисления. Нам не придется слишком беспокоиться об этом, но иногда это случается, и это полезно запомнить.

Например, следующая строка кода суммирует **0.1** и **0.1**. Вы могли бы ожидать, что результат будет **0.2**, но что вы на самом деле наблюдаете? Не беспокойтесь о странной строке в начале, она гарантирует, что результат будет напечатан с **20** цифрами после десятичного знака. Важно то, что суммирование находится в скобках:

```
print("%.20f" % (0.1 + 0.1))
```

Результат может отличаться на разных компьютерах — у меня эта строка выводит **0.20000000000000001000**.

И хотя разница между **0.2** и **0.20000000000000001** очень и очень мала, для компьютера это два разных числа.

Конечно, языки программирования также могут обрабатывать данные, которые не являются числами. Давайте посмотрим на текст.

Строки (Strings)

Ах, наш первый нечисловой тип данных. **Строки (Strings)** по сути являются текстом. Они называются строками, потому что представляют собой ряд (строку) символов. Мы уже имели дело со строкой в *Главе 1*, когда печатали *Hello, World* . Все, что находится в двойных (") или одинарных (') кавычках, является строкой. Вы можете сохранить строку в переменной следующим образом:

```
var character_name = "Erik"
```

Строки в GDScript могут содержать любой **символ Unicode**,

который по сути является любым символом из любого языка по всему миру и даже больше! Строки могут иметь любую длину, если только у вас не закончится память на компьютере.

Как уже упоминалось, вы можете использовать двойные или одинарные кавычки для создания строки, так как оба варианта будут работать. Просто убедитесь, что вы начинаете и заканчиваете строку одним и тем же типом кавычек. Соглашение в GDScript заключается в использовании двойных кавычек:

```
"Одна строка, заключенная в двойные кавычки, как это при
'Другая строка, с использованием одинарных кавычек'
```

Мы рассмотрели три основных типа данных в языках программирования. В следующем разделе мы поговорим об особом виде переменной, которая никогда не меняет своего значения – константе (constant).

Что такое константы?

Мы много говорили о переменных, но есть второй тип контейнера для размещения данных — **константы**. Константа — это особый вид переменной. Она также хранит любой тип данных, как и переменные, но вы не можете изменить её во время выполнения программы. Она остаётся постоянной. В остальном она точно такая же, как переменная, и имеет те же ограничения на своё имя.

Константы в GDScript

Давайте подробнее рассмотрим константу в скрипте. Чтобы определить константу, нужно использовать ключевое слово **const**, например:

```
const MAX_NUMBER_OF_BULLETS = 100
```

Обратите внимание, что имя этой константы — **MAX_NUMBER_OF_BULLETS**. В отличие от обычных переменных, для имён констант используются только заглавные буквы и подчёркивание. Это вопящий змеиный стиль **screaming snake case** или **SCREAMING_SNAKE_CASE**. Таким образом мы узнаём, что не сможем изменить их в нашем коде.

Использование константы похоже на использование переменной. Попробуйте это в вашем коде:

```
const FIREBALL_DAMAGE = 2
number_of_lives = number_of_lives - FIREBALL_DAMAGE
print("Количество жизней: ", number_of_lives)
```

Этот фрагмент кода должен вывести следующее:

```
Количество жизней: 3.
```

В качестве эксперимента попробуйте присвоить константе новое значение и посмотрите, какая ошибка возникнет:

```
const FIREBALL_DAMAGE = 2
FIREBALL_DAMAGE = 10
```

Интерпретатор выдаст следующую ошибку:

```
Cannot assign a new value to a constant
```

Это означает: «Невозможно присвоить новое значение константе».

Результат вы можете увидеть на *Рисунке 2.7*:

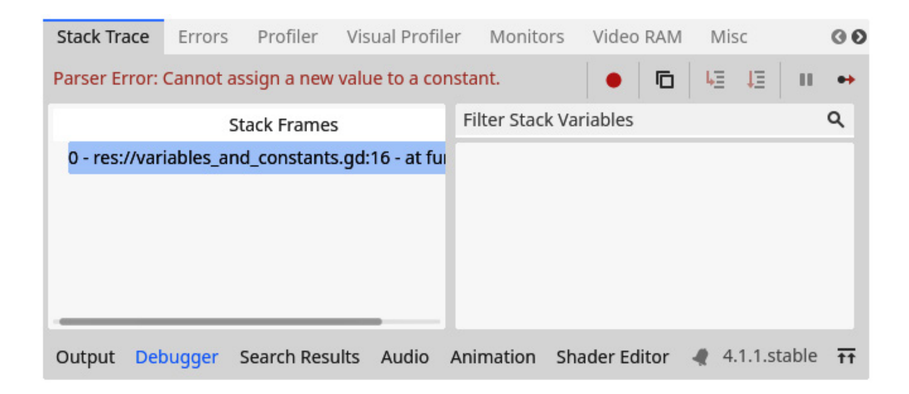


Рисунок 2.7 – Изменение константы приведёт к ошибке

Наряду с определением значений, которые не могут изменяться, константы также помогают нам, связывая имя со значением, чтобы нам не приходилось использовать необработанные значения. Имя константы помогает нам дать контекст тому, что представляет значение константы. Давайте теперь рассмотрим, почему это важно, с помощью магических чисел.

Магические числа

Константы используются для удаления **магических чисел**. В программировании мы называем число магическим числом, когда это просто число в коде без какого-либо контекста, например:

```
number_of_lives += 5
```

Из этого фрагмента кода вы не сможете понять, почему я добавил **5** к количеству жизней, но с правильно названной константой мы можем прояснить это, например, так:

```
const LIVES_RESTORED_BY_HEALTH_POTION = 5
number_of_lives += LIVES_RESTORED_BY_HEALTH_POTION
```

Магические числа, как правило, плохи и затрудняют чтение и

понимание кода.

Константы часто используются для поддержания чистоты кода; они показывают нам, какие значения никогда не должны меняться. Они также полезны во время отладки, потому что вы знаете, что они никогда не изменятся, и вам не нужно отслеживать, каково их значение на протяжении всего выполнения игры.

До сих пор мы рассмотрели все основные концепции, связанные с переменными и константами. Теперь давайте посмотрим, как мы можем создавать новые сцены, прежде чем погрузиться в потоки управления.

Создание новых сцен

Мы экспериментировали с одной и той же сценой и файлом скрипта. Работать таким образом было нормально, но нам нужен способ создания разных скриптов для разных тем и решений, чтобы разделить все уроки. Таким образом, нам не придется выбрасывать все наши предыдущие эксперименты и каждый раз очищать сцену.

К счастью, мы можем легко создать новую сцену и использовать другой скрипт. Мы даже можем дать имя сцене таким образом, чтобы можно было определить, что происходит внутри.

Важная заметка

Помните, что сцена — это отдельный файл, который может быть запущен. Он содержит набор узлов. На данный момент мы используем только один корневой узел. Позже, начиная с *Главы 6*, мы будем работать с более сложными сценами, которые содержат больше узлов.

Чтобы создать новую сцену, выполните следующие действия:

1. Щёлкните правой кнопкой мыши в любом месте окна **«Файловая система» (FileSystem)** и выберите **«Новая сцена...» (New Scene...)**.

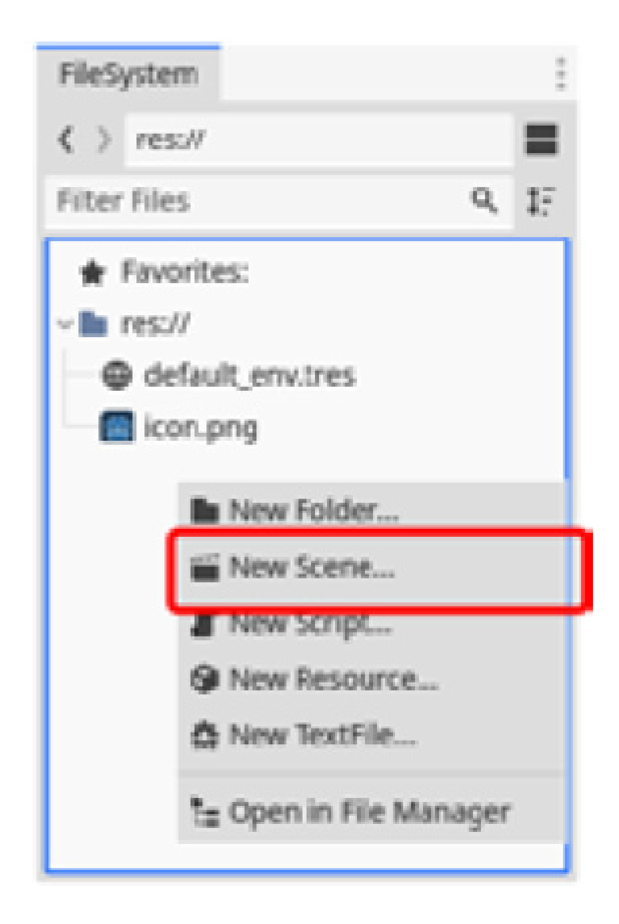


Рисунок 2.8 – Мы можем выбрать опцию создания новой сцены, щёлкнув правой кнопкой мыши в файловом менеджере.

1. Новая сцена загрузится в редактор.
2. Выберите «2D сцена», как мы это делали в *Главе 1*.
3. Дайте корневому узлу соответствующее имя, например, **ControlFlowTest**.
4. Сохраните сцену.
5. Прикрепите скрипт к корневому узлу, как в *Главе 1*. Вы можете оставить имя по умолчанию, сгенерированное из имени узла.

Вам следует повторять эти шаги для каждого упражнения, чтобы обеспечить упорядоченную структуру проекта. Хотите дополнительных плюсов? Создавайте папки для организации

кода и сцен. Подробнее об организации проектов в *Главе 5*.

Можно перемещать скрипты и сцены в разные каталоги. Просто убедитесь, что вы делаете это в доке **«Файловая система» (FileSystem)** самого Godot Engine. Таким образом, движок отслеживает, куда перемещаются файлы, и обеспечивает, чтобы все работало.

Если вы хотите запустить текущую открытую сцену, нажмите кнопку **«Запустить текущую сцену» (Run Current Scene)** (см. *Рисунок 2.9*). Обычная кнопка **Запустить (Run)** запускает главную сцену проекта, тогда как кнопка >**«Запустить текущую сцену» (Run Current Scene)** запускает текущую открытую сцену.



Рисунок 2.9 – Кнопка «Запустить текущую сцену» находится в правом верхнем углу

Хорошо, мы можем создать новую сцену для каждой главы или эксперимента. Давайте создадим новую сцену для следующего раздела и узнаем, как программы могут принимать решения с использованием потоков управления.

Начало работы с потоком управления

Сами по себе компьютеры довольно глупы. Они не знают, что делать или как рассуждать независимо. В этой части главы мы узнаем, как научить компьютер принимать решения на основе получаемых им данных.

Компьютеры делают именно то, что мы им говорим. Как вы поймете, изучая кодирование, это одновременно и благословение, и проклятие. Это означает, что мы должны быть

очень точными в формулировании того, чего мы от них хотим. К счастью, существует множество структур, которые предоставляют возможность составитьэти инструкции как можно точнее.

Оператор if

Самый простой способ принимать решения, на компьютере или нет, — использовать оператор **if**. Оператор **if** выглядит так: *Если здоровье игрока ниже нуля, закончить игру*. Мы могли бы обобщить это так: *если определенное условие истинно, выполнить действие*. Код для принятия этого решения может выглядеть так:

```
if number_of_lives < 0:
   print("Ты умер!")
```

Этот код можно прочитать как предложение: *если количество жизней меньше 0, вывести «Ты умер!»*. Рассмотрим отдельные части кода:

- Ключевое слово **if** указывает, что мы начинаем оператор **if**.
- Условие, в данном случае, это **numbers_of_lives < 0**.
- За условием следует двоеточие (**:**), указывающее начало блока кода, который будет выполнен *если (if)* условие истинно. В английском языке запятая часто выполняет разделение между ними.

Общая структура оператора **if** выглядит следующим образом:

```
if <condition>:
   <code block>
```

Обратите внимание, что блок кода имеет один отступ вправо с использованием символа табуляции.

Простейшие условия сравнивают два значения друг с другом. В GDScript операторы для сравнения значений перечислены в

Таблице 2.2:

Оператор	Описание	Пример	Результат
==	Равно (Equal)	2 == 2	Истина (true)
!=	Не равно (Not equal)	2 != 2	Ложь (false)
>	Больше (Greater than)	2 > 1	Истина (true)
<	Меньше (Smaller than)	3 < 1	Ложь (false)
>=	Больше или равно (Greater than or equal to)	1 >= 2	Ложь (false)
<=	Меньше или равно (Smaller than or equal to)	1 <= 2	Истина (true)

Таблица 2.2 – Операторы для сравнения значений в GDScript

Важная заметка

Оператор равенства использует два знака равенства (= =). Он будет интерпретироваться как оператор присваивания, если вы используете один знак равенства (=).

В качестве эксперимента попробуйте вывести результат следующего: **print(10 = = 5)** или **print(3 > = 1)**.

Если вы провели предыдущий эксперимент, вы бы увидели, что значения **true** и **false** были напечатаны в консоли. Они указывают, было ли условие оценено компьютером как **истинное** или **ложное**. Это новый тип данных, который мы называем **логическим (Boolean)**. Вы можете поместить Boolean в переменную, как и любой другой тип данных:

```
var player_died = number_of_lives < 0
var a_true_boolean = true
```

Переменная, содержащая логическое значение, может затем использоваться в операторе **if** в качестве замены всего условия:

```
if player_died:
   print("Ты мёртв!")
```

В отличие от других типов данных, которые могут иметь миллионы или миллиарды различных возможных значений, логические значения имеют только два возможных значения:

true или **false**. Вы также можете использовать эти значения напрямую, не сохраняя их.

В следующем коде мы видим логическое значение, используемое непосредственно в операторе **if** и как значение в переменной:

```
if true:
   print("Этот код всегда будет выполняться")
var always_false = false
if always_false:
   print("Этот код никогда не будет выполнен")
```

Мы узнали, что операторы **if** оценивают условия и выполняют свой блок кода только тогда, когда условие оказывается **истинным (true)**. Мы также узнали о логическом типе данных Boolean, который может иметь значение **true** или **false**. Но что, если мы хотим выполнить другой блок кода, если условие **ложно (false)**? Давайте рассмотрим это в следующем разделе.

Оператор if-else

Мы постоянно используем операторы **if** в повседневных разговорах. Расширением этой структуры, которое мы также используем очень часто, является оператор **if-else**. Допустим, игроку нужно набрать очки в игре, и в конце он выигрывает, если у него не менее **50** очков. *Если* счёт *игрока* больше или равен **50**, он выигрывает игру; в противном случае он проигрывает. Мы можем снова обобщить это так: *если определённое условие истинно, выполните одно действие; если это условие ложно, выполните другое действие*.

В GDScript мы выражаем оператор **if-else** следующим образом:

```
if score >= 50:
   print("Вы выиграли!")
else:
   print("Вы проиграли!")
```

Общая структура оператора **if-else** выглядит следующим образом:

```
if <условие>:
   <блок кода>
else:
   <блок кода>
```

Оператор **if** расширяется оператором **else** который находится на том же уровне, что и **if**. Обратите внимание на двоеточие (**:**) после ключевого слова **else**, а дальше — отступ, обозначающий начало нового блока кода.

Обратите внимание, что вы можете написать два оператора **if**, которые будут вести себя точно так же, как оператор **if-else**. Условие второго оператора **if** просто должно охватывать всё, что не охватывает первый:

```
if number_of_lives < 0:
   print("Ты умер!")
if number_of_lives >= 0:
   print("Ты жив!")
```

Обычно вы этого не делаете, поскольку стандартный код **if-else** имеет более простую структуру.

В качестве эксперимента создайте новую сцену и перепишите предыдущий фрагмент кода с оператором **if-else**.

Оператор elif

Иногда вам нужно больше, чем два разных результата. К счастью, мы можем расширить оператор **if** с помощью такого количества операторов **elif**, которое нам понадобится. **elif** — это сокращение от *else if* и функционирует почти так же, как обычный оператор **if**, за исключением того, что он всегда следует после оператора **if**:

```
if number_of_lives < 0:
   print("Ты умер!")
elif number_of_lives <= 1:
   print("Ты можешь умереть от одного удара!")
elif number_of_lives <= 3:
   print("У тебя осталось не так много жизней.")
else:
   print("Ты ЖИВ!")
```

Сначала компьютер оценит первый оператор **if**. Если он вернёт **true**, он выполнит свой код и остановится на этом. Если он вернёт **false**, он перейдет к следующему оператору **elif**. Если он вернёт **true**, он выполнит свой код и остановится на этом. Если он вернёт **false**, он перейдет к следующему **elif** и так далее. Если ни один из них не вернёт **true**, компьютер по умолчанию будет использовать код внутри оператора **else**, если таковой имеется.

Обратите внимание, что если вы проверили условие в предыдущем **elif**, вы можете быть уверены, что оно **ложно (false)** в последовательных условиях **elif**. Это потому, что если бы оно было **истинно (true)**, компьютер остановился бы там и выполнил бы блок кода этого **elif**. Поэтому вам не нужно снова перепроверять предыдущие условия.

Общая структура оператора **if-elif-else** выглядит следующим образом:

```
if <условие>:
   <блок кода>
elif <условие>:
   <блок кода>
<ещё операторы elif>
else:
   <блок кода>
```

На этом мы завершаем изучение различных операторов **if**. Теперь давайте посмотрим, как можно добавить комментарий к нашему коду, чтобы прояснить, чего мы пытаемся достичь.

Комментарии в коде

Как и написание книги, код может усложняться по мере увеличения его длины. Вдобавок ко всему, хотя вы можете понимать, что делает фрагмент кода во время его написания, вы можете забыть, что он на самом деле делает, возвращаясь к нему через недели или месяцы. Мы правильно называем наши переменные, что является одним из лучших способов борьбы с этой проблемой, но есть ещё один мощный метод — **комментарии**.

Комментарии — это текст в вашем коде, который не выполняется во время работы игры. Поэтому они предназначены исключительно для того, чтобы кто-то, читая код, понимал, что он делает.

Комментарии в GDScript начинаются со знака решетки (#). Всё, что находится после символа решетки, будет игнорироваться интерпретатором GDScript, например:

```
# Это комментарий, и он будет проигнорирован Godot
var number_of_lives = 2 # Это тоже комментарий
```

С этого момента я начну использовать комментарии в примерах кода, чтобы дать дополнительный контекст, и призываю вас делать то же самое! Я также буду использовать комментарии для обобщения и замены фактического кода, оставляя точную реализацию в качестве упражнения для вас.

Ещё одно интересное применение комментариев — комментирование определённых строк кода, когда они вам не нужны в данный момент, но вы не хотите полностью удалять их из кода. Это может выглядеть так:

```
# var number_of_lives = 2
var number_of_lives = 5 # Давайте попробуем 5 жизней вме
```

Сочетания клавиш для комментирования фрагментов кода в редакторе Godot — *Ctrl* + *K* для Windows и Linux и *Cmd* + *K*

для MacOS. Попробуйте выбрать код и использовать эти сочетания клавиш для комментирования строк здесь и там.

Важная заметка

Хотя комментарии — это очень полезно, всегда лучше называть переменные правильно, так что не пренебрегайте этим.

Отступы

Последовательные строки кода с одинаковым количеством пробелов слева принадлежат к одному и тому же блоку кода. Мы называем это пробельным **отступом**, потому что мы делаем отступ кода вправо.

В качестве эксперимента попробуйте не делать отступ в блоке кода после оператора **if** и посмотрите, какая ошибка возникнет.

Мы даже можем вкладывать блоки кода. Мы уже делаем это с функциями и операторами **if**, но также возможно вкладывать оператор **if** в другой оператор **if**, например:

```
var number_of_lives = 0
var damage_type = "fire"
if number_of_lives <= 0:
   print("Ты умер!")
   if damage_type == "fire":
      print("Сгорел в огне.")
   elif damage_type == "water":
      print("Утонул.")
```

Вы можете рассматривать отступы как пирамиду; каждый слой надстраивается над предыдущим, и каждый раз мы идем на один слой глубже. До сих пор мы видели, что функции и операторы **if** требуют блок кода, но вскоре мы увидим больше структур, которым он понадобится.

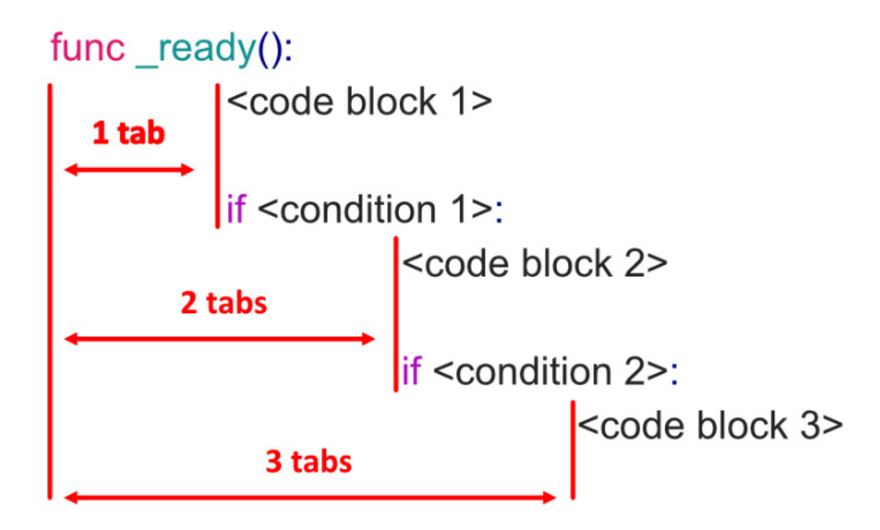


Рисунок 2.10 – Отступы разделяют различные блоки кода

Отступы в GDScript имеют решающее значение.

Важная заметка

Отступы также важны в Python — языке, который имеет много общего с GDScript. Большинство других языков программирования не особо заботятся об отступах, но они используют другие методы для разграничения блоков кода, такие как знаменитые фигурные скобки (**{}**).

Для каждого блока кода тип пробелов *и* их количество должны быть одинаковыми. Мы использовали одну табуляцию на слой отступа по мере углубления. Другой тип пробела, который вы можете использовать, — это обычные символы пробела, например, между словами предложения. Существует целый спор о том, что лучше — табы или пробелы. Мне нравятся табы, но если вы хотите использовать пробелы, используйте не менее четырёх из них, чтобы обозначить новый блок кода.

Важная заметка

Код в этой книге не использует вкладки по соображениям макета; вот почему я не рекомендую копировать и вставлять

код в вашу игру. Вы можете использовать полный код, доступный в репозитории GitHub книги здесь: https://github.com/PacktPublishing/Learning-GDScript-by-Developing-a-Game-with-Godot-4

В качестве эксперимента попробуйте использовать табы в одной строке и пробелы в другой строке кода того же блока кода, чтобы посмотреть, какая ошибка возникнет.

Если вы внимательно посмотрите на свой скрипт в редакторе скриптов, вы увидите, что он визуализирует отступы с помощью символов табуляции, как показано на *Рисунке 2.11*.

```
func _ready():
	var number_of_lives = 5
	
	if number_of_lives < 0:
		print("You died!")
	else:
		print("You live!")
```

Рисунок 2.11 – Редактор скриптов показывает табы в коде

В этой главе мы увидели, что GDScript ожидает, что блок кода будет присутствовать после оператора **if**. В *Главе 3* мы узнаем о большем количестве структур, которые требуют блоки кода.

Булева логика

Компьютеры не понимают двусмысленности. Человеческий язык, наоборот, очень двусмысленен. Мы можем эиспользовать это, потому что учитываем полный контекст разговора. Например, предложение «*Вы любите кофе или чай?*» можно интерпретировать двумя способами:

- Нравится ли вам один из них больше другого?
- Нравится ли вам один из них?

В зависимости от контекста, на этот вопрос могут быть даны ответы «*Чай*» или «*Кофе*», или же «*Да*» или «*Нет*». Однако компьютеры знают только то, что вы им говорите в данный момент времени. Поэтому компьютер предпочитает оперировать предложениями типа «*Выберите один вариант: а) Вы любите кофе; б) Вы любите чай*». Но как выразить всё это в коде?

Чтобы справиться с этим, умный математик придумал **Булеву логику**, названную в честь этого умного математика, Джорджа Буля. Этот небольшой, недвусмысленный язык упрощает общение с компьютерами. Он касается получения ответа *истина (true)* или *ложь (false)*из утверждений, как раз такого, который нам нужно вставить в выражения **если (if)**. Булева логика состоит из трёх основных компонентов:

- **Выражения**: Мы использовали их ранее, например, **number_of_lives == 2** или **5 < 1**.
- **Операторы**: Они объединяют операторы для создания более обширных операторов. Наиболее используемые операторы, которые нас интересуют, это **и (and)**, **или (or)**, а так же **не (not)**.
- **Скобки**: Как и в обычной математике, скобки изменяют порядок, в котором компьютер оценивает выражения.

Эти три элемента в сочетании очень мощные. Выражение в булевой логике выглядит так:

```
number_of_lives < 2 and is_hit
```

Здесь я использую оператор **and** для объединения двух операторов в один большой. Оператор вернёт **true** тогда и только тогда, когда оба меньших оператора, которые он объединяет, вернут **true** – то есть, если **number_of_lives** меньше **2** *и (and)* **is_hit** равен **true**. Во всех остальных случаях оператор вернёт **false**.

Другое утверждение в булевой логике может выглядеть так:

```
number_of_lives < 2 or damage >= 6
```

В этом выражении мы используем оператор **or**. Выражение вернет **true** если **number_of_lives** меньше **2** *или (or)* переменная damage больше или равна **6**. Таким образом, есть только один случай, когда выражение вернет **false —** когда number_of_lives равна или больше **2** *и* переменная damage меньше **6**.

Чтобы визуализировать эти отношения операторов, взгляните на следующие таблицы, называемые таблицами истинности. *Таблица 2.3* показывает результаты оператора **or** для всех возможных значений двух условий, которые он объединяет.

P or Q
Q Result
True
False True
False True
False

Table 2.3 – Возможные результаты оператора or

Таблица 2.4 показывает результаты оператора **and** для всех возможных значений двух условий, которые он объединяет.

P and Q
Q Result
True
False True
False True
False

Table 2.4 – Возможные результаты оператора and

Операторы **and** и **or** также можно объединить в один оператор, чтобы сделать его настолько длинным и сложным, насколько мы хотим:

```
number_of_lives < 2 and damage >= 6 or is_hit
```

В таком случае оператор **and** всегда будет оцениваться первым, а затем — оператор **or**. Это означает, что **number_of_lives < 2 and damage** будут оцениваться первыми, после чего результат будет объединён с **or is_hit**.

Чтобы изменить это поведение, вы можете использовать скобки. Например, если вы хотите сначала оценить **damage > = 6 or is_hit**, вы можете использовать скобки, например:

```
number_of_lives < 2 and (damage >= 6 or is_hit)
```

Последний интересующий нас оператор — это оператор **not**. Он отрицает утверждения, превращая **true** в **false** и наоборот, например:

```
not damage >= 6
```

Это выражение возвращает значение **true** если damage меньше **6** и **false**, если он больше или равен **6**. Обратите внимание, что оператор **not** не объединяет два выражения, а работает только с одним.

Вы также можете использовать **not** перед скобками, чтобы отрицать целую группу утверждений, например:

```
number_of_lives < 2 and not (damage >= 6 or is_hit)
```

Для визуализации результата действия оператора **not** смотрите следующую таблицу:

not P	**Result**
false	**true**
true	**false**

Table 2.5 – Возможные результаты оператора not

Важная заметка

В старых языках программирования, таких как C++ и Java, операторы **and**, **or** и **not** изображались разными символами (**&&**, ||, and **!**, соответственно). К счастью, в GDScript мы полностью их прописываем. Вы по-прежнему можете использовать символы в GDScript, если хотите, но запись операторов словами делает утверждения более лёгкими для чтения и понимания. Просто прочитайте утверждение, и оно будет иметь смысл в человеческом понимании, хотя и менее двусмысленно.

Имея за плечами булевую логику, давайте рассмотрим ещё несколько структур потока управления.

Выражение сопоставления match

В одном из предыдущих примеров мы использовали переменную **damage_type**, которая может содержать все различные виды повреждений в игре (огонь, вода, электричество и т.д.). Если бы мы хотели сделать что-то конкретное для каждого типа, мы могли бы использовать целый набор операторов **if-elif**, например:

```
if damage_type == "fire":
   # Делаем что-нибудь с огнём
elif damage_type == "water":
   # Делаем что-нибудь с водой
# И так далее …
else:
   # Тип повреждения не найден
```

Однако есть более приятный и мощный способ сделать это! Введём оператор **match**. Тот же фрагмент кода с использованием оператора **match** будет выглядеть так:

```
match damage_type:
   "fire":
```

```
        # Делаем что-нибудь с огнём
    "water":
        # Делаем что-нибудь с водой
# И так далее …
    _:
        # Тип повреждения не найден
```

В этом примере вы можете видеть, что мы проверяем, соответствует ли переменная **damage_type** любому из следующих значений. Если соответствует, мы выполняем блок кода под ним. Если ни одно из них не соответствует, мы выполняем код под случаем "_", который вы можете увидеть в блоке **else** в операторе **if-else**. Этот случай с подчёркиванием называется подстановочным знаком или случаем по умолчанию.

Общая структура оператора сопоставления **match** выглядит следующим образом:

```
match <переменная>:
    <шаблон>:
        <блок кода>
    <шаблон>:
        <блок кода>
    _:
        <блок кода>
```

Обратите внимание, что подчёркивание не является обязательным. При желании вы можете его не указывать, как мы это делаем в следующем примере.

Значения, с которыми мы сопоставляем, могут также быть другим типом данных, например числами. Мы даже можем иметь несколько разных значений для одного и того же случая, если разделим их запятой:

```
match a_variable:
    "fire", "water":
        # a_variable имела значение "огонь" или "вода"
```

```
   1, 2:
      # a_variable имел значение 1 или 2
```

В этом примере показано, что значения **fire** и **water** аставят оператор **match** выполнить первый блок кода, тогда как значения **1** и **2** заставят его выполнить второй блок.

Важная заметка

то не все типы шаблонов, с которыми может работать оператор **match**. Более полный список шаблонов для оператора **match** можно найти в официальной документации Godot по адресу https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_basics.html#match.

Операторы **match** и **if-elif-else** могут быть довольно большими. Давайте рассмотрим конструкцию, которая немного более компактна – оператор **ternary-if**.

Оператор ternary-if

Последняя структура потока управления в этой главе — это оператор **ternary-if**. Мы кратко коснемся его, поскольку стандартный оператор **if-else** — более читабельный способ записи того же поведения, но полезно нать, что такой тип потока управления существует.

Иногда вам может понадобиться сделать что-то вроде этого:

```
var amount_of_damage
if damage_type == "fire":
   amount_of_damage = 5
else:
   amount_of_damage = 1
```

Мы создаем переменную **amount_of_damage**, и присваиваем ей значение **5** если **damage_type** — "**fire**" и **1** во всех остальных случаях. Оператор **ternary-if** сжимает всю эту структуру в одну строку:

```
var amount_of_damage = 5 if damage_type == "fire" else 1
```

Эта строка будет иметь точно такой же эффект, как и предыдущий фрагмент кода. Он просто намного компактнее. Запутанная часть тернарного оператора **ternary-if** заключается в том, что он начинается со значения, которое возвращается, если условие истинно **(true)**. Вы должны читать это как «вернуть **5** если **damage_type** равен "**fire**", иначе вернуть **1**».

Общая структура оператора **ternery-if** выглядит следующим образом:

```
<value> if <condition> else <value>
```

Существует способ сцепить операторы **ternary-if** , чтобы создать что-то, что функционирует как структура **if - elif - else**. Однако я бы не советовал злоупотреблять этим и всегда использовал бы обычный оператор **if - elif - else** в таком сценарии, так как его легче читать и понимать.

Дополнительные упражнения – Заточка топора

Хотите дополнительной практики? Попробуйте следующие упражнения в **новой сцене и с новым скриптом**:

1. Возьмите следующий скрипт и добавьте оператор **if**, который проверяет, достаточно ли у игрока денег, чтобы купить предмет. Если да, используйте переменные, чтобы вывести «*Вы купили зелье за 10 золотых монет*». Если у игрока недостаточно денег, выведите «*У вас недостаточно денег*». Бонусные баллы, если вы вычислите и выведете новую сумму денег игрока после покупки предмета. Строка, которая говорит **randi() % 6 + 5**, просто генерирует случайное значение переменной **item_cost** между **5** и **10**, в которой хранится стоимость предмета:

```
extends Node
func _ready():
   var amount_of_player_money = 5 # Деньги игрока
   var item_cost = randi() % 6 + 5 # Стоимость пред
   var item_name = "Potion" # Название предмета
   # Ваш код
```

2. Завершите следующий скрипт всеми возможными комбинациями значений **true** и **false** для операторов **and**, **or** и **not**:

```
extends Node
func _ready():
   print(true and true)
   print(true and false)
   # Другие комбинации для оператора «and»
   print(true or true)
   # Другие комбинации для оператора «or»
   print(not true)
   # Другие комбинации для оператора «not»
```

3. Перепишите следующий фрагмент кода, чтобы использовать оператор **if-elif-else**:

```
var number_of_lives = 1
if number_of_lives >= 10:
   print("У вас полное здоровье")
if number_of_lives < 10 and number_of_lives > 2:
   print("У вас ещё осталось немного жизни")
if number_of_lives <= 2:
   print("У вас заканчивается здоровье")
```

4. Перепишите следующий фрагмент кода с оператором **match**:

```
var name = "Иван"
if name == "Эдик" or name == "Мария":
```

```
    print("Здравствуй ", name)
elif name == "Иван":
    print("Как дела, Иван?")
else:
    print("Привет, ", name)
```

Итоги

В этой главе мы изучили основные строительные блоки, на которых строятся все программы. Переменные необходимы для хранения и обработки данных, в то время как структуры потока управления дают нам возможность принимать решения на основе булевой логики.

В следующей главе мы расширим обе концепции. Мы рассмотрим более сложные типы переменных — массивы (arrays) и словари (dictionaries) и два новых типа потока управления (циклы **for** и **while**).

Опрос

- Как мы можем концептуально рассуждать о переменных в компьютерной системе?
- Какие из следующих имен переменных имеют описательное имя и отформатированы правильно?

  - ○ **var target_position**
  - ○ **var PlayerHealth**
  - ○ **var inventory**
  - ○ **var high score**
  - ○ **var x**
  - ○ **var enemy_name**

- Каковы результирующие значения следующих булевых выражений?

  - ○ **true and true**
  - ○ **false and true**

- ○ **false or not false**
- ○ **Not (true and false) and true**
- ○ **false or not 100 < 500**

- Что такое отступы и почему нам нужно делать отступы в коде в GDScript?
- Что выводит следующий фрагмент кода?

```
var number_of_lives = 0
Print("Вы живы" if number_of_lives > 0
```

3

Группировка информации в массивах (Arrays), циклах Loops) и словарях (Dictionaries)

В *Главе 2* мы узнали о важнейших основах программирования: переменных и потоке управления. Хотя эти строительные блоки могут показаться элементарными и ограниченными, они уже являются полными по Тьюрингу, то есть вы можете создать любую программу, которую когда-либо использовали с ними. Я не говорю, что вы должны или что это будет легко, но вы могли бы.

В этой главе мы узнаем о новых структурах данных и более продвинутых потоках управления, которые облегчат нам жизнь при работе с большими объемами данных. Сначала мы увидим, как массивы могут помочь нам создавать списки данных. Затем мы узнаем всё о циклах, очень мощной структуре потока управления для многократного выполнения блоков кода вместо одного. Наконец, мы узнаем о словарях, структуре данных, которая помогает нам группировать другие фрагменты данных в небольшие пакеты.

В этой главе мы рассмотрим следующие основные темы:

- Массивы (Arrays)
- Циклы (Loops)
- Словари (Dictionaries)
- Отладка (Debugging)
- Null

Технические требования

Если вы где-то застрянете, не забывайте, что вы можете найти пример всего, что мы делаем в этой главе, в папке **chapter03** репозитория. Репозиторий можно найти здесь: https://github.com/PacktPublishing/Learning-GDScript-by-Developing-a-Game-with-Godot-4/tree/main/chapter03.

Массивы (Arrays)

Часто мы хотим работать со списком данных, например, со списком предметов, которыми владеет игрок. **Массив** — это именно та структура данных, которую мы хотим использовать для таких случаев: это список, который может содержать элементы из других типов данных; это тип контейнера.

Контейнеры

Мы называем массив **контейнером**, что мы можем хранить и извлекать из него фрагменты данных других типов, например, целые числа, строки, логические значения и т.д. Массив содержит другие данные.

Контейнеры структурируют другие данные, чтобы с ними было легче работать.

Давайте рассмотрим, что такое массив в коде, как его создать и получить доступ к его элементам.

Создание массива

Создание массива выглядит так:

```
var inventory = ["Ключ", "Зелье", "Красный цветок", "Са
```

Здесь мы создали массив — список из четырех строк — и поместили его в переменную **inventory**. Обратите внимание,

что все элементы массива заключены в квадратные скобки **[]** и что каждый элемент отделен *запятой*.

Создание массива в одну строку, как мы только что сделали, часто нормально. Но иногда это может сделать строку кода слишком длинной. К счастью, мы также можем поместить каждый элемент в отдельную строку; это повышает читабельность и упрощает редактирование массива позже, если мы захотим добавить или удалить элемент:

```
var inventory = [
    "Ключ",
    "Зелье",
    "Красный цветок",
    "Сапоги"
]
```

Этот фрагмент кода создаст тот же массив, что и в начале этого раздела, но с дополнительным преимуществом: каждый элемент будет удобно размещён в новой строке, что облегчит чтение этого кода.

Важная заметка

Обратите внимание, что нам не нужно делать отступы для элементов массива с помощью табуляции, но общее соглашение заключается в том, чтобы делать это, и я призываю вас делать то же самое. Это часть философии чистого кода, которая делает более ясным, что эти элементы являются частью массива. Мы поговорим подробнее о написании чистого кода в *Главе 5*.

В качестве эксперимента попробуйте распечатать переменную, содержащую массив.

Доступ к значениям

Теперь, когда у нас есть наш список – массив – мы хотим иметь возможность доступа к его элементам. Для этого мы должны указать номер элемента, который мы хотим получить, в

квадратных скобках, сразу за именем переменной. Например, чтобы получить первый элемент нашего массива, мы должны написать следующее:

```
print( inventory[0] )
# Распечатывает: Ключ
```

Но что это? Я сказал вам, что мы извлекаем первый элемент массива, но я использовал для этого число 0. Это потому, что, в отличие от человеческого счета, массивы отсчитываются от 0, то есть они начинают отсчёт с 0. По сути, ко второму элементу можно получить доступ, используя **1** и так далее:

Содержание инвентаря	
Индекс	**Значение**
0	Ключ
1	Зелье
2	Красный цветок
3	Сапоги

Таблица 3.1 – Содержимое массива инвентаря

Счёт с нуля имеет большой смысл в контексте математических концепций и компьютерных алгоритмов, поэтому нам лучше к нему привыкнуть.

Мы называем позицию, которую занимает элемент в массиве, его **индексом (index)**.

В качестве эксперимента вместо прямого использования числа для извлечения элемента попробуйте поместить переменную, содержащую число, в квадратные скобки, например так:

```
var index = 3
print(inventory[index])
```

Вы также можете попробовать получить доступ к элементу, которого нет в массиве, например, к элементу **1000**, и посмотреть, какие ошибки всплывут. Попробуйте также

отрицательные числа.

Доступ к элементам в обратном направлении

Вы могли заметить что-то странное, если попробовали один из предыдущих экспериментов, где я просил вас получить доступ к элементу массива, используя отрицательные числа. Хотя отрицательные числа не всегда приводят к ошибке, некоторые возвращают элементы из массива.

Это потому, что если вы используете отрицательные числа, вы получаете доступ к элементам внутри массива с конца! Таким образом, элемент с индексом -1 является последним, -2 является предпоследним и т.д.:

```
var inventory = ["Ключ", "Зелье"]
prints("Последний предмет в вашем инвентаре — это: ", in
# Выведет: Зелье
```

Этот трюк с индексацией окажется очень полезным.

Изменение элементов массива

Интересное свойство массивов заключается в том, что мы можем рассматривать каждый элемент как обычную переменную. Чтобы присвоить новое значение элементу, например, мы можем просто присвоить ему новое значение:

```
inventory[3] = "Шлем"
```

Мы также можем использовать один из специальных операторов присваивания, например += или -=, о которых мы узнали в *Главе 2*, чтобы напрямую изменить одно из значений в массиве:

```
var array_of_numbers = [1, 4, -74, 0]
array_of_numbers[3] += 4
```

В предыдущем примере мы добавили **4** к третьему элементу нашего **array_of_numbers**, **0**, так что теперь значение равно **4**.

Использование оператора присваивания (=) позволяет легко изменять элементы в массиве.

Типы данных в массивах

Массивы могут содержать любой тип данных. Вы даже можете поместить несколько разных типов данных в один массив, например:

```
var an_array = [
   5,          # Целое число (Integer)
   "seven",    # Строка (string)
   8.9,        # Число с плавающей запятой (float)
   True        # Логическое значение (boolean)
]
```

Это плохая практика, потому что когда вы хотите получить доступ к одному из элементов, вы не знаете, с чем имеете дело. Вот почему я советую вам всегда использовать один тип данных для всех элементов в массиве.

Строки — это тайные массивы

Если вы вспомните *Главу 2*, когда я сказал, что строки называются так, потому что они являются *строками символов*, то это может не показаться большим сюрпризом. Но строку можно рассматривать как массив символов. Таким образом, мы можем получить один конкретный символ, так же как мы получили бы один конкретный элемент в массиве:

```
var player_name = "Эдик"
print(player_name[0])
```

```
# Выведет: Э
```

На практике мы используем это реже, но полезно знать, как работают строки «под капотом».

Манипулирование массивами

До сих пор мы создавали массивы, получали доступ к их элементам и даже изменяли эти элементы. Но есть еще много вещей, которые могут делать массивы. В отличие от стандартных типов данных, с которыми мы уже сталкивались, массивы предоставляют нам функции, которые мы могли бы использовать. Функции — это небольшие фрагменты кода, такие как функция **_ready()**, обеспечивавшая функциональность каждой сцены которую мы создавали. Эти функции могут что-то делать для нас.

Например, одна из этих функций может добавить дополнительный элемент в конец массива:

```
var inventory = ["Ключ", "Зелье"]
inventory.append("Меч")
```

Попробуйте распечатать эту переменную. Она покажет **[Key, Potion, Sword]** . Круто, правда?

Как видите, чтобы вызвать функцию массива, добавьте точку (.), а затем имя этой функции к имени этого массива. Это также применимо к другим типам данных.

Но что, если мы хотим добавить один массив к другому? Что ж, для этого тоже есть функция:

```
var loot = ["Золотая Монета", "Кинжал"]
inventory.append_array(loot)
```

Теперь весь массив **loot** будет добавлен в конец инвентаря.

Но подождите, это ещё не всё! А что, если вам нужно удалить

элемент? Вы используете функцию **remove_at()**. Эта функция удаляет элемент массива по определенному индексу:

```
inventory.remove_at(1)
```

Эта функция удалит элемент с индексом 1. Но что делать, если вы не знаете позицию, на которой находится элемент? Вы всегда можете ее найти!

```
var index_of_sword = inventory.find("Меч")
inventory.remove_at(index_of_sword)
```

Функция **find()** вернёт индекс элемента, который мы искали. Если она ничего не находит, она вернет число **-1**. Поэтому лучше всего проверить, равно ли возвращаемое ею число 0 или больше его. В противном случае вы можете удалить не тот элемент.

Функции **remove_at()** и **find()** сами по себе очень полезны, но есть также функция, которая объединяет их в одну! Это функция **erase()**, и её можно использовать следующим образом:

```
inventory.erase("Меч")
```

Эта строка кода даст тот же результат, что и фрагмент из двух предыдущих строк, поскольку она удаляет первый экземпляр слова "**Меч**", который она находит в массиве.

Массивы — важная концепция в программировании. Они динамически содержат произвольное количество элементов.

После небольшого отступления в сторону отладки в следующем разделе мы узнаем, как можно перебрать эти элементы и выполнить код для каждого из них отдельно, используя ключевые слова **for** и **while**.

Не бойтесь ошибок и предупреждений

Мы сталкивались с ошибками здесь и там при написании кода, особенно во время некоторых экспериментов. Эти ошибки часто содержат ценную информацию о проблеме и способах её решения. Давайте рассмотрим следующий фрагмент кода:

```
extends Node


func _ready():
	var 

Error at (5, 9): Expected variable name after "var".   1   5 :  9 | Tabs
```

Рисунок 3.1 – Ошибка сообщает нам, что за ключевым словом var должно следовать имя новой переменной

Здесь я остановился на полпути, определяя переменную. Редактор кода немедленно выдал мне ошибку. Как показано внизу, он сообщает мне, что ожидал имя переменной. Спасибо за подсказку, движок!

Давайте рассмотрим ошибку, которую движок не может предсказать до запуска кода:

```
func _ready():
   var inventory = [
      "Сапоги",
      "Бананы",
      "Пчелы"
   ]
   print(inventory[100])
```

На первый взгляд этот код может показаться правильным, но если вы поместите его в скрипт и запустите, то увидите следующее сообщение об ошибке:

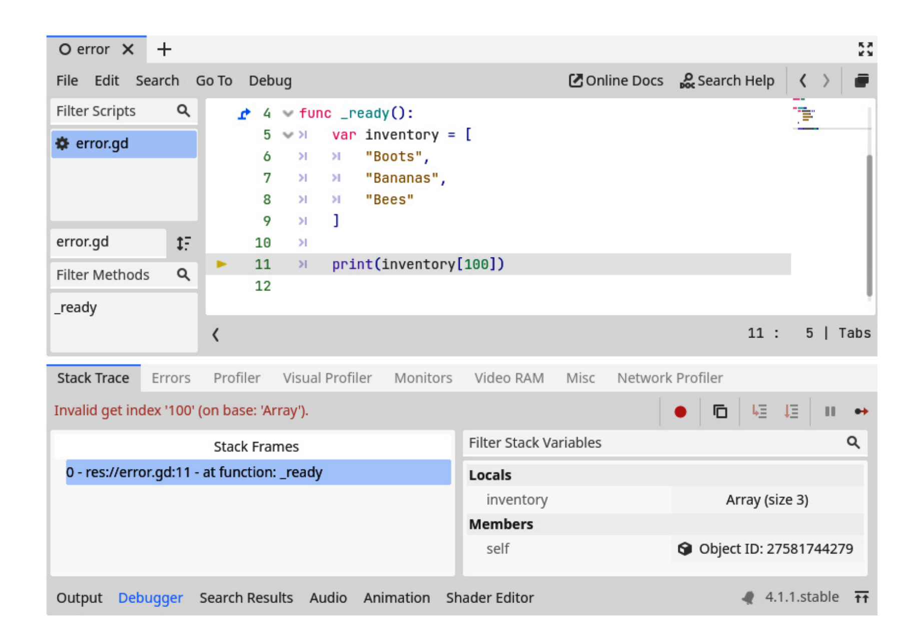


Рисунок 3.2 – После запуска этого кода интерпретатор предупреждает нас, что массив inventory не имеет 101-го элемента

Панель **Отладчик (Debugger)** открылась, чтобы чётко указать, что произошла ошибка *Invalid get index '100' (on base: 'Array')*. Это означает, что мы использовали недопустимый индекс для извлечения элемента из массива. Мы также видим жёлтую стрелку, указывающую точную строку, где произошла ошибка. Из этого можно сделать вывод, что массив inventory не имеет 101-го элемента, и поэтому нам не следует пытаться его распечатать.

Некоторые ошибки и предупреждения всплывают уже во время написания кода, например, первое, что мы увидели в этом разделе. Это позволяет легко гарантировать, что то, что мы пишем, будет запущено. К сожалению, написание кода, который не содержит ошибок или предупреждений на этом этапе, не гарантирует, что выполнение этого кода будет без ошибок и предупреждений. Это происходит потому, что с момента запуска фрагмента кода он может столкнуться со сценариями, которые анализатор кода никогда не мог бы рассмотреть.

Эти ошибки и предупреждения во время выполнения возникают, поскольку GDScript — слабо типизированный язык, что означает, что любая переменная может иметь любой тип и даже может изменять типы во время выполнения. Вот простой пример:

```
var my_vairable = 5
my_variable = "Привет, мир!"
```

Но это означает, что во время выполнения часть кода может дать сбой. В *Главе 4* вы узнаете, как справиться с этой неопределенностью элегантным способом, сохраняющим гибкость слабо типизированных переменных.

Цените предупреждения и ошибки. Движок показывает их не для того, чтобы запугать нас, а чтобы подтолкнуть нас в правильном направлении для создания лучшего, более надёжного программного обеспечения. Если бы движок не заботился об этом и не останавливал игру при возникновении ошибки, то мы могли бы выпустить полусломанную игру. Это не то, чего мы хотим! Мы хотим чтобы игроки получали удовольствие от игры и наименьшее количество ошибок!

Баги

Когда игра каким-то образом сломана, будь то сбой или логическая ошибка, мы говорим, что в ней есть **баги**. Этот термин появился в одном из первых случаев, когда компьютер дал сбой, и виновником оказался настоящий жук (bug), заползший в машину. Но этот термин использовался и до этого такими людьми, как Томас Эдисон, для описания «небольших неисправностей и трудностей» в аппаратной части.

В следующем разделе мы узнаем о циклах.

Циклы

Мы помещали строки в коде одну за другой, и Godot Engine прекрасно их выполнял сверху вниз. Но наступает момент,

когда мы хотим повторить одну или несколько строк кода. Например, что, если мы хотим распечатать каждый предмет в инвентаре игрока с некоторым форматированием?

Мы могли бы сделать что-то подобное с помощью следующего кода:

```
var inventory = ["Ботинки", "Бананы", "Бинты"]
for item in inventory:
   print("У вас есть ", item)
```

Это называется **циклом**. Этот конкретный цикл называется **цикл for**. В следующих нескольких разделах мы рассмотрим два вида циклов, которые присутствуют в GDScript.

Циклы for

Цикл **for** будет повторять свой блок кода для каждого элемента массива. В случае примера, приведенного во введении, этот элемент будет доступен в переменной **item**, которая действительна только в пределах цикла **for**. Конечно, мы можем использовать ту же структуру для других массивов и называть временную переменную по-другому. Я выбрал имя *item (предмет)*, потому что именно предметы содержит *inventory (инвентарь)*. Другим подходящим именем может быть *item_name (имя_предмета)*.

Мы можем визуализировать цикл **for** с помощью следующей блок-схемы:

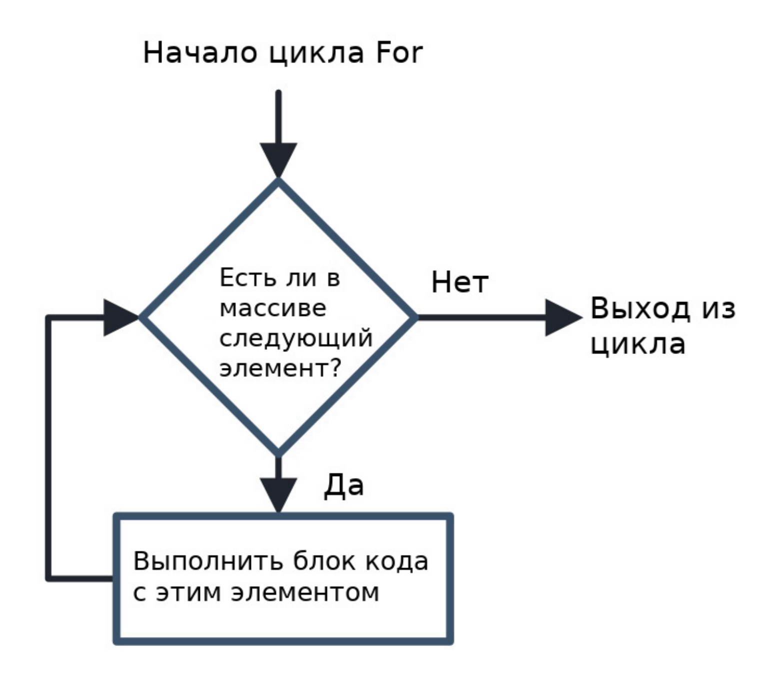


Рисунок 3.3 – Поток цикла for во время выполнения кода

Как показано на *Рисунке 3.3*, мы начинаем цикл сверху. Если есть первый элемент, мы следуем по стрелке *Да* и выполняем блок кода внутри цикла для этого элемента. Затем мы проверяем, есть ли следующий элемент. Если да, мы снова выполняем блок кода, теперь для этого элемента, и так далее, пока не останется больше элементов.

Синтаксис, то есть общая структура, цикла **for** выглядит следующим образом:

```
for <имя_временной_переменной> in <имя_массива>:
    <блок_кода>
```

Синтаксис

Предшествующая структура также называется синтаксисом языка. Она фиксирует правила того, что возможно и как вещи

должны быть определены в языке.

Цикл **for** — очень мощная структура потока управления. Давайте рассмотрим некоторые другие варианты использования, в которых мы можем его применить.

Функция диапазона (range)

Иногда вам понадобится перебрать все индексы в массиве. Для этого вы можете использовать функцию **range()**, встроенную в движок. Эта функция возвращает массив от **0** до указанного числа. Вот пример:

```
var numbers_from_0_to_5 = range(6)
print(numbers_from_0_to_5 )
```

Этот код выведет **[0, 1, 2, 3, 4, 5]** .

Обратите внимание, что мы передали функции число **6** , но массив останавливается на числе 5.

Мы можем использовать функцию **range()** следующим образом:

```
var inventory = ["Ботинки", "Бананы", "Бинты"]
for index in range(inventory.size()):
   print("Элемент по индексу ", index, " содержит ", inv
```

Здесь я вызвал функцию **size()** для массива **inventory**. Она возвращает размер массива, который затем мы можем напрямую подключить к функции **range()**.

Функция **range()** может даже больше. Если вы предоставите ей два числа, она создаст массив, который начнется с первого числа и дойдет до второго, снова исключая его. Попробуйте это:

```
for number in range(10, 20):
   print(number)
```

В качестве эксперимента попробуйте составить цикл **for** с предыдущей функцией **range()** и вывести результат, то есть **range(16, 26, 2)**.

Вы увидите, что мы переходим от **16** к **26**, как и ожидалось. Но на этот раз интервал между каждым числом равен **2**. Таким образом, мы должны получить числа **16 , 18 , 20, 22** и **24**.

Третий аргумент, передаваемый команде range, определяет размер шага между числами.

Теперь, когда у нас есть цикл **for** и функция **range()**, давайте рассмотрим цикл while.

Циклы While

Второй тип цикла в GDScript — это **цикл while**. Этот цикл работает немного похоже на оператор **if —** мы задаем ему условие и повторяем его блок кода, пока условие оценивается как **true**. Вот пример:

```
var inventory = ["Ботинки", "Бананы", "Бинты", "Краги",
while inventory.size() > 3:
   inventory.remove_at(0)
```

Здесь мы удаляем первый элемент массива inventory такой длины, чтобы длина массива составляла более трёх элементов.

Синтаксис цикла **while** выглядит следующим образом:

```
while <condition>:
   <code_block>
```

Когда Godot сталкивается с оператором **while**, он выполняет следующие шаги:

1. Сначала он оценивает условие. Если оно истинно (true), он переходит к *шагу 2*; в противном случае он полностью пропускает блок кода и переходит к *шагу 4*.

2. Далее будет выполнен блок кода.
3. Затем он вернётся к *шагу 1*, чтобы снова оценить состояние и посмотреть, изменилось ли оно.
4. Наконец, он выполнит оставшуюся часть кода.

Таким образом, **циклы while** будут выполняться до тех пор, пока определённое нами условие возвращает значение **истина (true)**.

Мы можем визуализировать цикл **while** с помощью следующей блок-схемы:

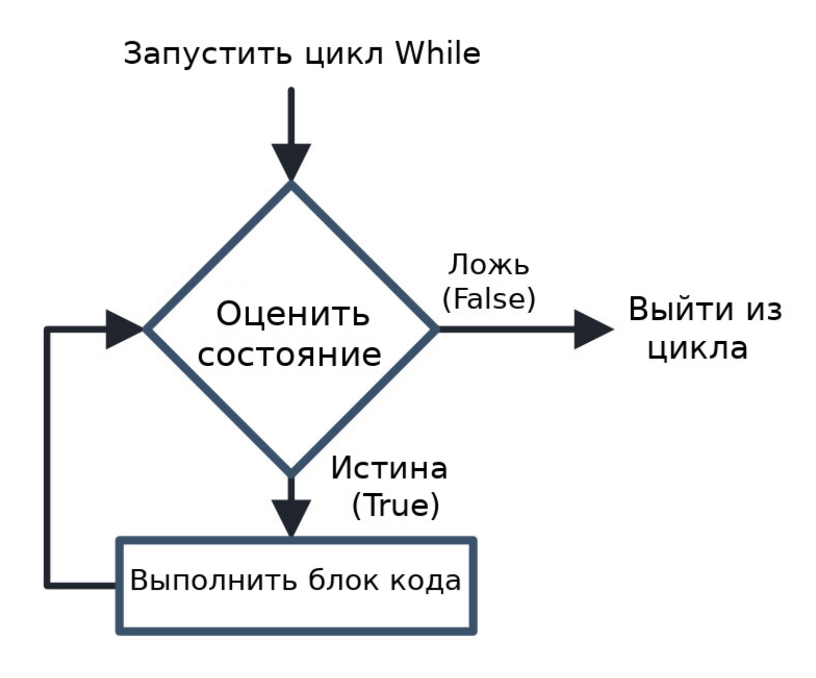


Рисунок 3.4 – Поток цикла while во время выполнения кода

На *Рисунке 3.4*, мы начинаем цикл, оценивая указанное нами условие. Если результатом является значение **true**, мы выполняем блок кода цикла. Если это значение не оценивается как **true**, мы выходим из цикла.

Бесконечные циклы

Блок кода в операторе **while** должен позаботиться о том, чтобы условие стало ложным (false). Иначе возникнет бесконечный **цикл**. К счастью, Godot предотвращает сбои компьютера, но это может вызвать зависание вашей игры и её крах.

Теперь, когда мы узнали о двух основных циклах в GDScript, давайте посмотрим, как мы можем получить больше контроля над этими циклами с помощью некоторых специальных ключевых слов, которые мы можем использовать только внутри этих циклов.

Продолжение или разрыв цикла

Два ключевых слова могут использоваться только в цикле: **continue** и **break**. Чрезмерное использование или злоупотребление ими не является лучшей практикой; вы можете избежать обоих, если правильно построите свой цикл. Но их всё равно необходимо знать.

Ключевое слово continue

Ключевое слово **continue** может использоваться в цикле для прямого перехода к началу. В цикле **for** это означает, что мы переходим к следующему элементу массива. В цикле **while** это означает, что мы возвращаемся к оценке условия:

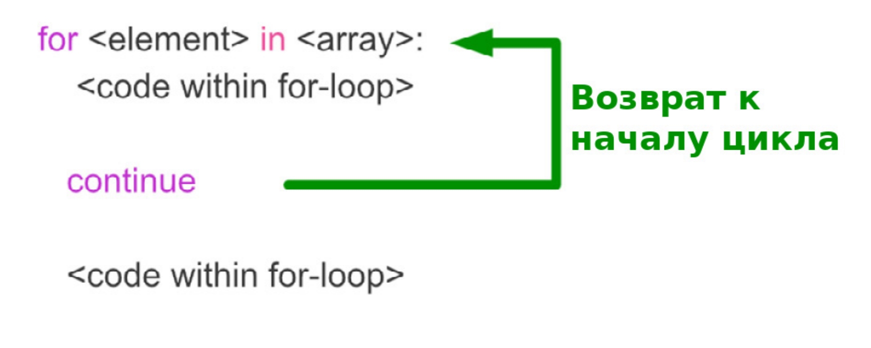


Рисунок 3.5 – Ключевое слово continue пропустит весь последующий код и вернёт поток управления к началу цикла

для следующего элемента в массиве.

Например, следующий код выведет на экран все предметы в инвентаре, но пропустит **Банан**:

```
var inventory = ["Ботинки", "Банан", "Бинты"]
for item in inventory.size():
   if item == "Банан":
      continue
   print(item)
```

Результат этого цикла будет следующим:

```
Ботинки
Бинты
```

Ключевое слово **continue** весьма полезно, когда вы хотите пропустить элементы, но что, если вы хотите полностью остановить выполнение цикла? Вот тут-то и приходит на помощь ключевое слово **break**. Давайте рассмотрим это сейчас.

Ключевое слово break

Иногда вам захочется преждевременно остановить выполнение цикла. В таком случае мы можем использовать ключевое слово **break**:

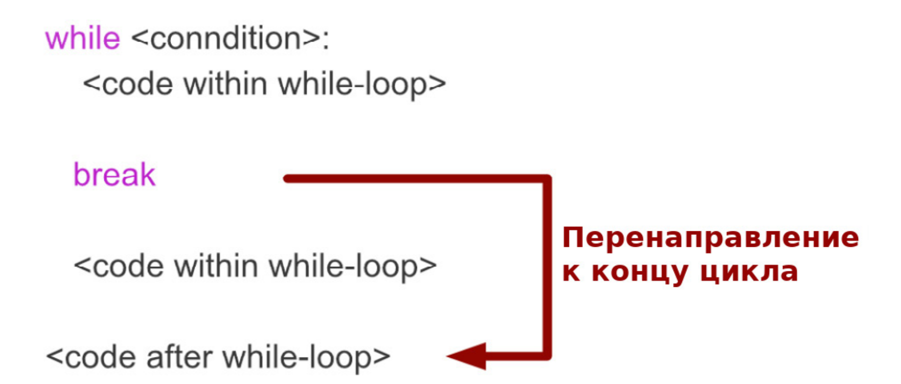


Рисунок 3.6 – Ключевое слово break пропустит весь

последующий код и остановит выполнение цикла

Интерпретатор прекратит всё, что он делал в цикле, и перейдёт к коду после него. Вот пример:

```
var inventory = ["Ботинки", "Бинты", "Бананы", "Краги",
while inventory.size() > 3:
   if inventory[0] == "Бананы":
      break
   prints("Удаляем ", inventory[0])
   inventory.remove_at(0)
```

Этот фрагмент выведет на экран следующее:

```
Удаляем Ботинки
Удаляем Бинты
```

Затем он увидит, что первым предметом в инвентаре являются **Бананы**, поэтому условие **if** оценивается как **истина (true)**, и мы прерываем цикл, полностью останавливая его.

Итак, мы увидели, как использовать **continue** и **break**, но как я уже говорил ранее, можно написать тот же цикл без этих ключевых слов. В качестве эксперимента попробуйте переписать оба примера, использующие ключеве слова **continue** и **break** так, чтобы они имели одинаковое поведение, но не используйте ключевые слова **continue** или **break**.

Циклы позволяют нам запускать код для неопределенного количества элементов в массиве, делая наш код более гибким и динамичным. Мы будем использовать их на протяжении всей этой книги. Теперь давайте рассмотрим другой тип данных контейнера: словари (dictionaries).

Словари

Словарь (dictionary) это еще один контейнер данных, как и массив. Но в отличие от массивов, которые хранят данные в

определённом порядке, , словари хранят данные с помощью пары **ключ-значение (key-value)**. Вместо того чтобы связывать каждый элемент с предопределенным числом, как в массиве, мы связываем их с ключом, который определяем сами. Поскольку мы должны определять собственные ключи, в словаре более жёсткая структура, чем в массиве.

Создание словаря

Допустим, мы хотим сохранить название, цену и вес предмета в нашей игре. Мы можем сделать это с помощью словаря:

```
var item = {
   "name": "Ботинки",
   "price": 5,
   "weight": 3.9
}
```

Здесь мы используем фигурные скобки **{}**, для определения словаря. Затем мы определяем ключ и связанное с ним значение внутри фигурных скобок. Например, ключ "**name**" связан со значением "**Boots**". Каждая пара ключ-значение должна быть отделена запятой:

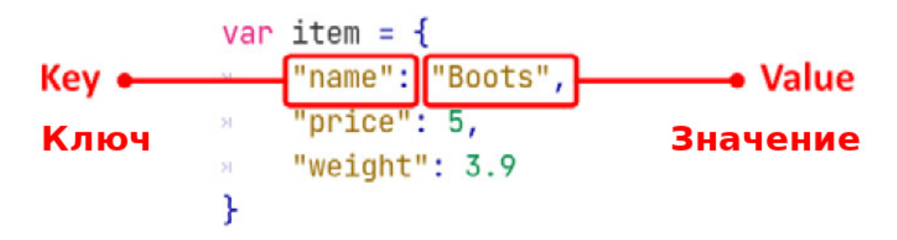


Рисунок 3.7 – Словари состоят из одной или нескольких пар ключ-значение

В качестве эксперимента попробуйте распечатать предыдущий словарь следующим образом:

```
print({
```

```
    "name": "Ботинки",
    "price": 5,
    "weight": 3.9
})
```

Словари помогают нам организовать данные в более структурированном виде. Теперь давайте посмотрим, какие данные мы можем поместить в словарь.

Типы данных в словарях

Какие типы данных мы можем использовать для ключей и значений?

Значения в словаре могут быть любого типа данных, даже массивами или другими словарями!

Однако, на ключи это не распространяется. Мы можем использовать только простые типы данных в качестве ключей в словаре. Строки, числа с плавающей точкой и целые числа допустимы. Более сложные типы данных, такие как массивы или словари, не допускаются.

Стоит отметить, что тип ключей и значений не обязательно должен быть одинаковым во всем словаре. Вы можете использовать разные типы данных повсюду:

```
var a_mess_dictionary = {
    "string_key": [4, 6, 9],
    3.14: "Pi",
    123: {
        "sub_key": "Это подсловарь"
    }
}
```

Как видите, словари are ver — очень мощные структуры для организации данных.

Доступ к значениям и их изменение

Доступ к значениям словаря и их изменение очень похожи на то, как мы получаем доступ к значениям массива. Вместо указания индекса элемента в квадратных скобках мы указываем ключ нужного нам значения:

```
var item = {
   "name": "Ботинки",
   "price": 5,
   "weight": 3.9
}
print(item["name"])
item["price"] += 10
```

Мы даже можем использовать сохранённый ключ из переменной:

```
var key_variable = "name"
print(item[key_variable])
```

Наконец, если ключ, к которому вы пытаетесь получить доступ, является строкой, вы также можете получить его значение, используя следующий синтаксис:

```
print(item.name)
```

Просто используйте точку после имени словаря, а затем ключ. Это не сработает, если ключ — это что-то, кроме строки, например, число.

Создание новой пары ключ-значение

Замечательная особенность словарей заключается в том, что мы можем легко добавлять новые пары ключ-значение после создания словаря. Мы можем просто присвоить значение несуществующему ключу:

```
item["color"] = "blue"
```

Здесь мы добавили новую пару ключ-значение в словарь элементов, где "**color**" является ключом, а "**blue**" — значением.

Полезные функции

Как и массивы, словари имеют ряд полезных функций, которые иногда оказываются полезными.

has

Иногда нам нужно узнать, содержит ли словарь определенный ключ. В таком случае мы можем рассчитывать на функцию **has()**:

```
var item = { "name": "Банан" }
if item.has("name"):
   print(item.name)
```

Поскольку в словаре **item** есть ключ с именем *name*, этот код выведет это имя.

erase

Мы только что увидели, как добавлять пары ключ-значение в словарь. Но с помощью **erase()** мы также можем удалить пару:

```
var item = { "name": "Банан" }
item.erase("name")
```

В конце этого фрагмента словарь элементов будет пуст.

В качестве эксперимента попробуйте распечатать словарь элементов после того, как сотрёте все ключи.

Обход циклом по словарям

Это может gjrfpfnmcz удивительнhsv, но вы можете проходить по

словарям, как и по массивам. Например, если мы хотим распечатать всю информацию в словаре item, мы могли бы сделать что-то вроде этого:

```
var item = {
   "name": "Ботинки",
   "price": 5,
   "weight": 3.9
}
for key in item:
   prints(key, "содержит", item[key])
```

Как видите, временная переменная **key** переносит ключи словаря один за другим.

Мы также можем напрямую перебрать все значения словаря вместо того, чтобы сначала получать ключи:

```
for value in item.values():
   print(value)
```

Этот цикл выведет все значения в словаре.

Важное примечание

Обход циклом по массиву или другой структуре данных также можно назвать *итерацией* по нему.

В качестве эксперимента попробуйте распечатать значения словаря с помощью **item.values()**: **print(item.values())**

Вложенные циклы

Если вы хотите получить больше удовольствия от циклов, вы можете вкладывать их. Под этим мы подразумеваем, что вы можете использовать цикл внутри другого цикла. Например, предположим, что у нас есть инвентарь, который является массивом словарей предметов, и мы хотим распечатать информацию о каждом предмете в удобном виде. Мы могли бы

сделать что-то вроде этого:

```
var inventory = [
   {
      "name": "Ботинки",
      "price": 5,
   },
   {
      "name": "Волшебные перчатки",
      "price": 10
   },
   {
      "name": "Крутые очки",
      "price": 58
   }
]
for item_index in inventory.size():
   print("Параметры элемента ", item_index, ":")
   var item = inventory[item_index]
   for key in item:
     printt(key, item[key])
```

Сначала мы перебираем все элементы массива, что даёт нам каждый элемент словаря. Затем мы перебираем все ключи этого элемента и выводим ключ и его значение.

Конечно, вы также можете свободно комбинировать циклы **while** и **for**. Ограничений нет!

Итак, мы узнали все о двух основных потоках управления циклами и двух основных типах данных контейнера в GDScript. В следующем разделе мы узнаем о новом типе данных: **null**.

Null

Наконец, позвольте мне представить вам новый тип данных: **null**. Этот тип имеет только одно значение: **null**. Он не несёт никакой информации, не может быть изменён и не имеет

функций, которые вы можете вызвать. Итак, для чего он хорош? Это значение переменной, когда мы не задаём ей её значение самого начала. Попробуйте следующий фрагмент кода:

```
var inventory
print(inventory)
```

Вы увидите, что он выведет **null**. Иногда вам захочется сделать это, чтобы убедиться, что переменная существует, но вы пока не хотите инициировать её со значением. В метафоре картотечного шкафа из *Главы 2* это означало бы, что мы зарезервировали ящик и имя для переменной, но пока не заполнили его данными.

Использование переменной любым способом, пока она имеет значение null, приведёт к ошибке. Например, следующие две операции приведут к ошибке при выполнении кода:

```
var inventory
inventory.append("Ботинки")
var number_of_lives
number_of_lives -= 2
```

Поэтому лучше всего проверить, является ли переменная **null**, если вы не уверены, что переменная инициализирована:

```
var number_of_lives
if number_of_lives != null:
   number_of_lives -= 2
```

Некоторые операторы или функции возвращают **null**, когда они не могут выполнить свою задачу, как ожидалось. Например, если вы обращаетесь в словаре к ключу, который не существует, он вернёт значение **null**:

```
var item = {
   "name": "Ботинки",
```

```
    "price": 5,
    "weight": 3.9
}
print(item["height"])
```

В предыдущем примере будет выведено значение **null**.

Дополнительные упражнения – Заточка топора

1. Напишите скрипт, который находит и выводит название самого дорогого предмета в следующем массиве с помощью цикла **for**. Вам нужно будет сохранить две переменные **most_expensive_item** и **max_price**. Переменная **max_price** начинается с 0. Теперь, каждый раз, когда вы сталкиваетесь с более дорогим предметом, вы сохраняете этот предмет в переменной **most_expensive_item** и обновляете значение переменной **max_price**. Самым дорогим предметом в следующем массиве должно быть *Ring of Might*:

   ```
   var inventory = [
       { "name": "Банан", "price": 5 },
       { "name": "Кольцо Могущества", "price": 100 },
       { "name": "Зелье исцеления", "price": 58 },
       { "name": "Шлем", "price": 44 },
   ]
   ```

2. Напишите скрипт, который проверяет, является ли конкретная строка палиндромом; это означает, что строка должна выглядеть одинаково, читаете ли вы ее вперед или назад. Например, *rotator* является палиндромом, а *bee* — нет. Для этого вам придется перебирать строку в двух направлениях одновременно.

Итоги

В этой главе мы рассмотрели два новых типа контейнеров, массивы и словари, и два типа циклов — **for** и **while**. Мы также узнали о полезных функциях в типе данных string и познакомились со значением **null**.

В следующей главе мы узнаем всё о классах — пользовательских типах данных, которые мы можем определять самостоятельно.

Опрос

- О каких двух типах контейнеров мы узнали в этой главе?
- В чём разница между массивами и словарями?
- Как получить доступ к четвертому значению в следующем массиве?

```
var grocery list = ["Яблоки", "Мука", "Салат", "Жел
```

- Как получить доступ к значению height в следующем словаре?

```
var person = {
    "name": "Майк",
    "eye_color": "карий",
    "hair_color": "блондин",
    "height": 184,
}
```

- Что возвращает **range(2, 9)**?
- В чем разница между циклами **for** и **while**?
- Когда мы используем один цикл внутри другого цикла, называем ли мы это вложенным циклом?
- Какое значение имеет следующая переменная?

```
var number_of_lives
```

4

Создание структуры с помощью методов и классов

В *Главе 3* мы узнали о типах коллекций и циклах. Эти мощные концепции помогли нам структурировать наши данные и запускать код произвольное количество раз.

Возможность повторно использовать код в цикле — это здорово, но что, если мы хотим повторно использовать этот код в любой произвольный момент времени? А что, если мы хотим повторно использовать целые структуры кода и данных, такие как, например, враги или транспортные средства?

Методы и классы — это именно те концепции, которые помогут нам достичь этого уровня повторного использования!

В ходе этой главы мы рассмотрим последние несколько основных концепций программирования. К концу мы узнаем всё, что нужно, чтобы называть себя настоящими программистами.

В этой главе мы рассмотрим следующие основные темы:

- Функции
- Классы
- Указание типа
- Объектно-ориентированное программирование (ООП)

Технические требования

Если вы где-то застрянете, не забывайте, что вы можете найти пример всего, что мы делаем в этой главе, в папке **chapter04** репозитория. Репозиторий можно найти здесь: https://

github.com/PacktPublishing/Learning-GDScript-by-Developing-a-Game-with-Godot-4/tree/main/chapter04.

Методы — это фрагменты кода, которые можно использовать повторно.

В *Главе 1* мы научились писать код с помощью метода **_ready()**, принадлежащего узлу. Мы увидели, что код, содержащийся в этой функции, будет выполняться с момента начала работы нашей игры. Теперь давайте подробнее рассмотрим, что такое функции и как их можно использовать.

Метод против функции

Термины *method* и *function* часто используются как взаимозаменяемые. Они обозначают два очень похожих понятия, но применяются по-разному. В этой книге мы будем использовать оба как взаимозаменяемые.

Что такое функция?

Функция объединяет блок кода в единое целое, чтобы мы могли его повторно использовать без необходимости переписывать тот же код. Мы уже использовали функции повсюду. Например, чтобы найти индекс элемента в массиве, мы использовали функцию **find()**:

```
var inventory = ["Амулет", "Бананы", "Свечи"]
print(inventory.find("Бананы"))
```

Под капотом интерпретатор ищет блок кода, связанный с функцией **find**, выполняет его со строкой **Bananas** в качестве входных данных, а затем возвращает нам результат.

В предыдущем случае мы распечатали результат. Обратите внимание, что оператор **print**, который мы использовали в этом коде, также является просто функцией!

Входные данные, которые мы передаем функции, называются **аргументами**.

Если максимально упростить технические аспекты, то функция — это всего лишь обходной путь, который наша программа совершает, отклоняясь от своего обычного пути выполнения — обходной путь через другой блок кода.

Определение функции

Давайте рассмотрим функцию, которая снижает здоровье игрока:

```
func lower_player_health(amount):
   player_health -= amount
```

Как видите, для определения функции нам понадобятся следующие части:

- Ключевое слово **func**. Оно указывает GDScript, что мы собираемся определить новую функцию, подобно тому, как ключевое слово **var** используется для объявления переменных.
- Имя. Это имя, которое мы будем использовать для вызова функции, в данном случае **lower_player_health()**. Убедитесь, что вы выбрали описательное имя, как и в случае с именами переменных.
- Список параметров, разделённых запятыми и заключенных в скобки. В нашем случае есть только один параметр: **amount**. Это те самые данные, которые мы хотим, чтобы пользователь функции предоставил ей. Наличие каких-либо параметров не является обязательным.
- Блок кода, который выполняется, когда мы вызываем функцию. В этом блоке кода мы можем использовать

параметры функции, как если бы они были обычными переменными.

Аргументы и параметры

Внимательные читатели могли заметить, что когда мы вызываем функцию, входные данные называются **аргументами**, а внутри функции мы называем их **параметрами**. Параметры — это, по сути, входные переменные функции, тогда как аргументы — это конкретные значения, с которыми мы вызываем функцию.

Но не бойтесь путать терминологию; это делает почти каждый программист, и все поймут, о чем вы говорите.

Но не бойтесь путать терминологию; это делает почти каждый программист, и все поймут, о чем вы говорите.

```
func <function_name>(<parameter1>, <parameter2>):
   <code_block>
```

Стоит отметить, что количество определяемых параметров может варьироваться. В примере синтаксиса мы определили два параметра, но могли бы определить сотню или ни одного.

В качестве примера приведём функцию, которая просто выводит **Привет, мир**:

```
func say_hello():
   print("Привет, мир")
```

Присвоение имени функции

Имена функций имеют те же ограничения, что и имена переменных:

- Они содержат только буквенно-цифровые символы.
- Пробелов быть не должно.
- Они не могут начинаться с цифры.

- Их не следует называть по аналогии с существующими ключевыми словами.

Но в отличие от имен переменных, важно, чтобы имя функции отражало то, что делает код внутри функции. Таким образом, вы знаете, чего ожидать при запуске функции.

Вот несколько примеров хороших названий функций:

```
calculate_player_health()
apply_velocity()
prepare_race()
```

А вот несколько примеров плохих названий функций:

```
do_the_thing()
calculate()
a()
```

Именование функций, как и именование чего угодно в программировании, занятие сложное, но важное. Необходимо давать всему понятные описательные имена.

Ключевое слово keyword

В циклах **for** и **while** мы использовали ключевое слово **break** для преждевременного выхода из цикла. В функциях у нас есть очень похожее ключевое слово: **return**. Это ключевое слово заставит поток выполнения немедленно выйти из функции. И справедливости ради, если вы поместите оператор **return** в цикл, он также остановит этот цикл, потому что мы больше не выполняем функцию в целом.

Поместите его в любое место функции, и мы сможем вернуться туда, где вызвали функцию, даже если это означает, что определённый код никогда не будет выполнен:

```
func a_cool_function():
```

```
    print("Этот фрагмент кода будет выполнен")
    return
    print("Этот фрагмент кода НИКОГДА не будет выполнен")
```

Функции также могут возвращать значения, как мы видели на примере функции **find()** для массивов, которая возвращала индекс искомого нами значения. Чтобы вернуть значение, мы снова используем ключевое слово **return**, но на этот раз мы указываем значение, которое хотим вернуть, сразу после него:

```
func minimum(number1, number2):
    if number1 < number2:
        return number1
    else:
        return number2
```

Теперь мы могли бы использовать эту функцию **minimum()**, чтобы получить наименьшее из двух значений:

```
print(minimum(5, 2))
var lowest_number = minimum(1, 300)
```

Выполнение этого фрагмента кода выведет число **2** и заполнит переменную **lowest_number** числом **1**.

В этом разделе мы реализовали нашу собственную функцию **minimum()**, но эта функция на самом деле уже существует в движке и называется **min()**. Так что с этого момента вы можете использовать ту, которую предоставляет движок, чтобы найти наименьшее число.

Ключевое слово pass

При создании нового скрипта мы уже видели функцию **_ready()**, структурированную следующим образом:

```
func _ready():
    pass
```

Это фактически пустая функция, ожидающая заполнения программистом. Она ничего не делает. Но нам все еще нужен блок кода внутри функции; в противном случае движок выдаст ошибку. Вот тут-то и появляется ключевое слово **pass**. Это строка кода, которая вообще ничего не делает. Таким образом, мы можем использовать ее для создания блока кода, который не несет никакой логики. Таким образом, мы можем создавать пустые функции.

Пустые функции очень полезны в ООП, о чем мы поговорим в *Главе 5*.

Необязательные параметры

Чтобы сделать функцию более гибкой, вы можете решить указать некоторые параметры как необязательные. Таким образом, вы сможете позже решить, предоставлять аргументы или нет. Для этого мы должны предоставить значение по умолчанию для этого аргумента.

Если вы не укажете значения для этих параметров при вызове функции, GDScript примет указанные нами значения по умолчанию.

Мы можем использовать эту технику для расширения нашей предыдущей функции удаления жизни из общего количества здоровья игрока:

```
extends Node
var player_health = 2
func lower_player_health(amount = 1):
   player_health -= amount
```

В предыдущем примере функция **lower_player_health()** имеет один параметр, **amount**, который является необязательным. Мы знаем, что он необязателен, потому что мы задаем ему значение по умолчанию в определении с помощью знака равенства. Если мы вызовем эту функцию и дадим ей аргумент, она будет использовать этот аргумент для заполнения суммы.

Если мы не дадим ей никакого аргумента, она будет использовать значение по умолчанию **1** в качестве значения суммы. Мы можем использовать эту функцию следующим образом:

```
lower_player_health(5) # Вычтет 5 из здоровья игрока
lower_player_health(2) # Вычтет 2 из здоровья игрока
lower_player_health() # Вычтет 1 из здоровья игрока
```

Если функция имеет несколько параметров, из которых один или несколько являются необязательными, необязательные параметры всегда должны быть последними в определении. Это связано с тем, что если вы пропустите один из аргументов, GDScript не сможет угадать, какой из них, и просто предположит, что он последний. Если мы случайно перепутаем порядок параметров, мы получим ошибку от редактора кода, который сообщит нам, что нужно правильно их упорядочить.

Допустим, нам нужно написать функцию, которая перемещает игрока под определенным углом с определенной скоростью, а также указать, бежит ли игрок и может ли он сталкиваться с объектами в мире:

```
func move_player(angle, is_running, speed = 20, can_coll
   # тело функции
```

Эту функцию **move_player()** можно использовать более разнообразными способами, чем функцию **lower_player_health()**:

```
move_player(.5, true) # Не заполнен ни один из необязате
move_player(.5, true, 100) # Заполнен один из необязател
move_player(.5, true, 1, false) # Заполнены два необязат
```

Как видите, мы можем выбирать, какие необязательные параметры заполнять, при условии, что мы всегда указываем их в том порядке, в котором они были объявлены в определении функции.

Функции являются основой всего программирования. Многие программы работают только с теми типами данных, о которых мы узнали до сих пор, и функциями. Но давайте сделаем еще один шаг вперед и узнаем, как можно сгруппировать данные и функции в одну связную единицу с помощью классов.

Классы группируют код и данные вместе

Наконец, мы добрались до одной из важнейших революций в компьютерной науке, которая потрясла мир языков программирования в середине 60-х: **классов**.

Некоторые умные компьютерные инженеры задумались о том, как мы используем данные и функции, и увидели, что мы часто используем выбранный набор функций на выбранном наборе данных. Это привело их к объединению этих функций и данных вместе, чтобы они жили очень тесно друг с другом. Такая группа называется классом.

В играх классы часто моделируют отдельные сущности. Мы могли бы иметь класс для следующих сущностей:

- Игрок
- Враги
- Подбираемые предметы
- Препятствия

Каждый из этих классов содержит и управляет своими собственными данными. Класс игрока может управлять здоровьем и инвентарём игрока, в то время как подбираемые предметы управляют тем, какие свойства у них самих и какое влияние они оказывают на игрока.

По сути, каждый класс — это пользовательский тип данных, как и те, что мы видели раньше. Но теперь мы сами создаём данные и функции! Это очень мощная концепция, так что давайте начнём!

Определение класса

Чтобы создать простой класс, мы просто используем ключевое слово **class** с именем, которое мы хотели бы дать этому классу. После этого мы можем начать собирать класс, определяя переменные и методы, которые он охватывает:

```
class Enemy: # класс врага
   var damage = 5 # урон
   var health = 10 # здоровье
   func take_damage(amount): # нанесение урона
      health -= amount
      if health <= 0:
         die()
   func die(): # гибель
      print("Аааргх я умер!")
```

Здесь мы видим класс именем **Enemy**; он имеет две переменные-члена — **damage** и **health**, и два метода-члена — **take_damage()** и **die()**.

Создание экземпляра класса (Instancing)

Вы можете рассматривать класс как чертёж или шаблон нашего пользовательского типа данных. Итак, как только у нас есть класс с определенными переменными-членами и функциями, мы можем создать из него новый экземпляр. Мы называем этот экземпляр **объектом**. Каждый объект существует сам по себе. Это означает, что между ними нет общих данных. Чтобы создать новый объект, мы указываем имя класса и вызываем для него функцию **.new()**:

```
var enemy = Enemy.new()
```

Теперь эта переменная содержит объект нашего собственного класса **Enemy**! С помощью этого объекта мы можем получить

доступ к его переменным-членам и вызывать его функции:

```
print(enemy.damage)
enemy.take_damage(20)
```

Мы также можем использовать этот объект в типах контейнеров, таких как массивы и словари, и передавать его в качестве аргумента функциям:

```
var list_of_enemies = [
   Enemy.new(),
   Enemy.new(),
]
var dict_of_enemies = {
   "Enemy1": Enemy.new(),
}

var enemy = Enemy.new()
any_function(enemy)
```

Вы видите, что экземпляры класса можно использовать так же, как и любые другие виды переменных.

Присвоение имени классу

Классам нужны имена, как и переменным и методам. Хотя имя класса имеет те же ограничения, что и имя переменной, принято склеивать слова в имени друг с другом и писать первую букву каждого с заглавной буквы. Мы называем это **Pascal case** или **PascalCase**, потому что оно было популяризировано в Pascal, языке программирования, созданном в 1970 -х годах. Вот несколько примеров:

```
Enemy #Враг
HealthTracker # Индикатор здоровья
InventoryItem # Предмет инвентаря
```

Это всё отличные имена классов. В *Главе 5* мы рассмотрим ещё несколько советов по именованию классов.

Расширение класса

Мы также можем создать новый класс, расширив уже существующий. Это называется **наследованием (inheritance)**, потому что мы наследуем все данные и логику из родительского класса в дочерний класс и расширяем его новыми данными и логикой.

Например, чтобы создать нового врага на основе предыдущего, мы следуем этой структуре:

```
class BuffEnemy extends Enemy:
   func _init():
      health = 100
   func die():
      print("Как ты меня победил?!?")
```

Вы можете видеть, что после имени нового класса мы указываем ключевое слово **extends**, а затем класс, от которого хотим унаследоваться. Чтобы перезаписать переменные исходного класса, нам нужно установить их в функции **_init()**. Это специальная функция, называемая **конструктором**, которая вызывается с момента создания объекта класса **BuffEnemy**. Конструктор должен инициализировать объект, чтобы он был готов к использованию.

Вы также можете видеть, что мы можем переопределить методы, поскольку я перезаписываю функцию **die**, чтобы вывести другую строку. Когда класс**BuffEnemy** получает урон и умирает, он вызовет функцию **die** унаследованного класса, а не родительского.

Если мы создадим объект класса **BuffEnemy**, то увидим, что его здоровье действительно равно **100** и он не умрет от **20** единиц урона, а когда враг умрет, он выведет новую строку из перезаписанной функции:

```
var buff_enemy = BuffEnemy.new()
print(buff_enemy.damage)
buff_enemy.take_damage(20)
print(buff_enemy.health)
buff_enemy.take_damage(80)
```

В качестве эксперимента попробуйте создать нового врага, самостоятельно расширив класс **Enemy**.

Каждый скрипт — класс!

Я открою вам небольшой секрет. Каждый скрипт, который мы написали до сих пор, уже является классом! Вы, возможно, уже поняли это после прочтения раздела *Расширение класса*, потому что первая строка каждого скрипта, который мы написали, была для расширения класса **Node**! Этот класс является базовым классом для каждого типа узла в движке Godot.

Этот класс **Node** содержит кучу шаблонных данных и кода, которые нужны Godot для использования во время игры. Большая часть этого не представляет для нас интереса в данный момент. Но некоторые из них представляют интерес, включая следующее:

- **Методы жизненного цикла**: это методы, которые выполняются в определенные моменты жизненного цикла узла, например, при его создании, уничтожении или обновлении.
- **Дочерние и родительские узлы**: в Godot узлы следуют иерархической структуре, и каждый узел имеет ссылку на свои дочерние и родительские узлы. Наличие доступа к ним очень помогает при работе с заданной иерархической структурой.

Узел, к которому мы прикрепляем скрипт, связывается с этим скриптом, а значит, с данными и логикой скрипта и по сути является экземпляром объекта скрипта.

В *Главе 7* мы научимся расширять узлы на конкретных

примерах **Node2D** и **Sprite**.

В то время как обычные классы должны иметь имя, класс, полученный из скрипта, не имеет имени, хотя это возможно. Просто используйте ключевое слово **class_name** в верхней части скрипта:

```
class_name MyCustomNode
extends Node
# Остальная часть класса
```

Godot облегчает нам создание нового класса.

Когда доступны определённые переменные?

Вы, возможно, уже заметили, что переменные, которые мы определяем, доступны не отовсюду. Каждая переменная имеет определенную область, в пределах которой вы можете её использовать. Давайте подробнее рассмотрим следующий фрагмент кода:

```
func _ready():
   var player_health = 5
   if player_health > 2:
      var damage = 2
   player_health -= damage
```

Если вы введете этот код в редакторе скриптов, что я вам и рекомендую сделать, то в последней строке вы увидите сообщение об ошибке, сообщающее, что переменная **damage** не находится в области видимости (scope). Это означает, что переменная нам недоступна, и мы не можем ее использовать.

В целом существует пять вариантов, при которых нам доступна переменная:

- Переменная была определена в том же блоке кода, где мы

ее используем, например:

```
var player_health = 2
print(player_health)
```

- Переменная была определена в блоке кода, который является родительским для текущего блока кода, например:

```
var player_health = 2
if player_health > 1:
   print(player_health)
```

- Переменная была определена в текущем классе следующим образом:

```
extends Node
var player_health = 2
func _ready():
   print(player_health)
```

- Переменная была определена глобально. Мы узнаем больше об этом типе переменных в *Главе 10*. Но достаточно сказать, что этот тип переменной доступен в любом месте в любое время, в любом скрипте, даже в самом редакторе. Этот тип переменных очень полезен для хранения информации, используемой многими различными процессами. Мы называем их **автозагрузками (autoloads)**.
- Переменная была встроена в движок. Эти переменные доступны нам на глобальном уровне; мы не определяли их сами. Список этих глобальных переменных, констант и функций можно найти здесь: https://docs.godotengine.org/en/stable/classes/class_%40globalscope.html.

Вот несколько примеров:

```
PI # одержит константу числа Пи, около 3,1415
Time
OS
```

Область, в которой доступна переменная, называется её **областью действия (scope)**.

Хотя невозможно определить две переменные с одинаковым именем в пределах одной области действия, можно определить две переменные с одинаковым именем, когда одна находится вне текущей функции, а другая внутри неё. Мы называем это **затенением (shadowing)** , потому что одна находится в тени другой. Например, одна переменная определена внутри класса как переменная-член, а другая — внутри функции, например:

```
extends Node
var damage = 3
func a_function():
   var damage = 100
   print(damage)
```

Если вы запустите предыдущий код, вы увидите, что он выведет **100**, поскольку в случае сомнений GDScript всегда будет брать ближайшую определённую переменную:



Рисунок 4.1 – Предупреждение, сообщающее нам, что переменная damage определена в скрипте и в функции

Однако, как показано на *Рисунке 4.1*, вы также увидите, что движок выдает предупреждение о двойном использовании имени переменной, что может привести к путанице для нас, разработчиков.

Область действия функции

Функции также имеют определённую область действия, хотя и немного более ограниченную, чем область действия переменной. Вы можете использовать функцию в следующих вариантах:

- Функция была определена в классе, который мы используем.
- Функция была определена внутри любого родительского класса, от которого наследуется наш класс.
- Функция была встроена в движок. Такие функции доступны отовсюду. Вот пример:

```
print("Эй")
max(5, 3) # Возвращает наибольшее из двух чисел
sin(PI) # Возвращает значение синуса для угла
```

Но, как мы видели ранее, мы также можем вызвать функцию класса объекта. Таким образом, область действия этой функции будет такой же большой, как и область действия объекта.

Типы помогают нам узнать, как использовать переменную.

Мы видели разные типы данных и даже знаем, как создавать свои собственные. Но была одна большая проблема! Переменные могли менять тип в процессе выполнения. Это особенно раздражает, потому что если мы используем неправильный тип данных в определённойситуации, игра зависнет!

```
var number_of_lives = 5
number_of_lives += 1
number_of_lives = {
   player_lives = 5,
```

```
   enemie_lives = 1,
}
number_of_lives += 1
```

В предыдущем коде первая отмеченная строка будет работать, но вторая приводит к сбою игры! Этот сбой происходит потому, что в первом случае мы добавляем **1** к значению **5**, другому числу, а во втором случае мы пытаемся добавить **1** ко всему словарю. Эта операция не поддерживается и, таким образом, приводит к сбою игры.

К счастью, есть способ, которым мы можем использовать наши знания о том, какой тип данных мы ожидаем для определённых операций или функций. Это то, что мы узнаем в ходе этого раздела.

Что такое указание типа?

Другие популярные языки, такие как C++, C#, Java, Golang и Rust, решают проблему незнания типа данных переменной, неявно указывая, какой тип она будет иметь с момента её определения. Не существует (почти) способа определить переменную, не закрепив за ней определенный тип. Кроме того, в других языках тип переменной, в отличие от GDScript, не может быть изменён в процессе работы программы.

В GDScript тоже есть система, которая делает что-то подобное, но менее ограничительная. Эта система называется **указанием типа (type hinting)**, потому что мы даем указание о том, какой тип мы хотели бы видеть у переменной. Это помогает GDScript заранее определить, сработает ли операция или приведёт к сбою игры.

Давайте рассмотрим различные способы указания типа в GDScript.

Указание типа переменных

Например, если мы хотим указать, что количество жизней

игрока всегда будет целым числом, то есть *integer*, мы можем указать тип этой переменной, например, так:

```
var number_of_lives: int = 5
```

То же самое можно сделать и для других типов данных:

```
var player_name: String = "Эрик"
var inventory: Array = ["Бокалы с прохладительным", "Нап
```

Если мы попытаемся присвоить значение другого типа переменной с указанным типом, как в следующем примере, редактор кода выдаст нам предупреждение перед запуском игры и ошибку во время её запуска:

```
var inventory: Array = ["Бокалы с прохладительным", "Нап
inventory = 100
```

Обратите внимание, что мы можем указать тип переменной только при её определении. После определения мы можем свободно использовать переменную, и движок должен знать, является ли она переменной с определённым типом. Вот почему мы не можем позже просто добавить тип или изменить его.

Указания типов помогают нам выявлять ошибки до того, как они произойдут!

Указание типа массивов

Помимо указания того, что определённая переменная является типом **Array**, мы также можем указать тип значений, которые мы можем найти в этом массиве. Это очень полезно и позволяет нам легко узнать, какие данные ожидать в массиве.

Чтобы указать, какие типы данных могут быть найдены в массиве, просто укажите этот тип в квадратных скобках после типа **Array**, например:

```
var cool_numbers: Array[float] = [3.1415, 6.282, 2.71828
```

В предыдущем фрагменте явно указано, что **cool_numbers** — это массив чисел с плавающей точкой, и поэтому каждый элемент этого массива следует рассматривать как число с плавающей точкой.

В качестве эксперимента попробуйте следующую строку кода. Будет ошибка. Но почему?

```
var inventory: Array[String] = ["Бокалы с прохладительны
```

Если вы попробуете, то увидите, что это приведёт к ошибке, поскольку мы даём указание, что переменная **inventory** — это массив, заполненный строками. Но одно из значений в массиве — число. Движок увидит это и выдаст ошибку.

Изучение типа Variant

В фоновом режиме GDScript будет использовать **Variant** как почти любой тип переменной. Класс **Variant** может содержать почти любой другой тип данных. Вот почему мы можем переключать тип переменной в процессе выполнения, когда не указываем тип при её создании.

Переменные, для которых мы указали тип, тоже являются типом **Variant**. Но к ним прикреплены дополнительные требования к типу, например, их значение должно быть целым числом или словарём.

В GDScript мы никогда не имеем дело напрямую с функциональностью класса **Variant**. GDScript аккуратно оборачивает его вокруг любого значения, которое мы ему назначаем, и поэтому нам не нужно беспокоиться о типе **Variant**. Мы можем просто рассуждать о типе данных, которые мы храним в переменной.

Указание типа параметров функции

Помимо указания типа переменной, мы также можем указать тип параметров функции. Делается это таким образом:

```
func take_damage(amount: int):
   player_health -= amount
```

Теперь, если вы попытаетесь вызвать эту функцию с аргументом, который не является целым числом, редактор предупредит вас, что вы совершаете ошибку. Например, взгляните на следующую строку кода, которая использует функцию **take_damage()** из предыдущего фрагмента кода:

```
take_damage("Два")
```

В этом случае движок выдаст ошибку, поскольку функция **take_damage()** ожидает целочисленное значение, а строка несовместима с целым числом.

Автоматическое преобразование переменных

Когда вы попробуете выполнить **take_damage(1.5)**, вы увидите, что редактор не выдаёт предупреждение или ошибку. Это происходит потому, что GDScript автоматически преобразует определённые переменные из одного типа в другой. Это называется **неявным преобразованием (implicit conversion)**.

Одно из таких преобразований происходит между числами с плавающей точкой и целыми числами. В этом случае GDScript округлит число с плавающей точкой до ближайшего целого значения. Для нашего небольшого примера это означает, что **1.5** будет округлено до целого значения **1**.

Указание типа можно также объединить со значением по умолчанию для параметра; просто укажите тип первым, а значение по умолчанию — после:

```
func take_damage(amount: int = 1):
   player_health -= amount
```

Функция **take_damage()** теперь принимает один параметр, **amount**, который имеет тип integer и значение по умолчанию **1**.

Указание типа для возвращаемого значения функции

Мы также можем указать тип значения, которое будет возвращено функцией. Это очень полезно, поскольку даёт нам много информации о том, чего ожидать от этой функции. Это можно сделать так:

```
func minimum(number_1: float, number_2: float) -> float:
   if number_1 < number_2:
      return number_1
   else:
      return number_2
```

Эта функция **minimum()** всегда должна будет возвращать число с плавающей точкой, независимо от того, какой оператор **return** это делает.

В качестве эксперимента попробуйте ничего не возвращать в функции, тип которой предполагает возврат числа с плавающей точкой. Вы увидите, что движок выдаст нам ошибку.

Использование void в качестве возвращаемого значения функции

Иногда функция вообще не возвращает значение. В этом случае мы можем указать возвращаемое значение этой функции, используя тип **void**. **void** нельзя использовать для переменных. Он используется только в определениях функций. Таким образом, **void** указывает, что функция ничего не возвращает:

```
var player_health: int = 5
func subtract_amount_from_health(amount: int) -> void:
```

```
    player_health -= amount
```

Однако большинство людей опускают указание типа **void**, когда функция ничего не возвращает, и только указывают тип функции, когда она действительно что-то возвращает. Хорошо знать, что указание типа **void** существует, когда вы с ней где-то сталкиваетесь.

Выведенные типы

Есть второй способ типизации переменной без явного указания типа. Этот метод использует распознавание типов самого движка. Мы можем использовать тип первого значения, назначенного переменной, как тип этой переменной для остальной части выполнения. Мы можем сделать это следующим образом:

```
var number_of_lives := 5
```

Это выглядит очень похоже на обычное, нетипизированное определение переменной. Но на этот раз мы ставим двоеточие перед знаком равенства. Это зафиксирует тип переменной как соответствующий тип значения, которое мы ей присваиваем.

Этот метод называется **выводом типа (type inferring)**, поскольку в GDScript переменная просто принимает тип значения, которое мы передаём ей во время назначения.

Обратите внимание, что, как и в случае с обычным указанием типа переменной, мы можем вывести тип переменной только при её определении. Поэтому следующий код работать не будет:

```
var number_of_lives
number_of_lives := 5 # This will error
```

Выведение типа может облегчить нам указание типа переменных без необходимости заранее думать о фактическом типе.

null может быть любого типа

Знание того, какому типу принадлежит переменная, не означает, что нам не придётся искать переменные, которые являются **null**. **null** может быть присвоен переменной любого типа, который не является базовым типом (**int**, **float**, **String**, и т.д.). Таким образом, массивы, словари, самоопределяемые классы, и т.п. могут по-прежнему быть **null**, если они не инициализированы:

```
var inventory: Array[String] = ["Бананы", "Зола", "Драко
inventory = null # Это законно
inventory.find("Дракон") # Это приведёт к сбою игры
```

null часто используется для сброса переменных в пустое состояние.

Автодополнение

Ещё одним большим преимуществом типизации наших переменных является то, что текстовый редактор поможет нам, когда мы захотим вызвать функцию или перейти к переменной-члену класса, нам подскажет автодополнение. Например, если у нас есть строка и мы начинаем вводить, чтобы вызвать функцию для неё, небольшое всплывающее окно покажет все возможные функции, к которым мы пытаемся получить доступ. Затем мы можем просто продолжать вводить или использовать клавиши со стрелками, чтобы выбрать нужную функцию, и нажать *Enter*, чтобы выбрать одну из них. Это очень помогает, если вы знаете, что хотите сделать, но не совсем уверены, как была вызвана функция, или просто для ускорения ввода длинных имён функций:

```
var inventory: Array = ["Buckle", "Uranium"]
inventory.fi
        fill
        filter
        find
        rfind
```

Рисунок 4.2 – Редактор кода поможет нам с автодополнением при использовании подсказок типов

Autocomplete — наш общий друг, поэтому если сделать его более полным, то в долгосрочной перспективе это нам только поможет.

Использование подсказок типов для именованных классов

В дополнение к встроенным типам мы также можем типизировать наши собственные пользовательские классы. Но для этого нам сначала нужно зарегистрировать имя для типа нашего класса. Чтобы зарегистрировать имя, мы можем использовать ключевое слово **class_name**, за которым следует имя, которое мы хотели бы, чтобы тип данных нашего класса имел в верхней части файла, например так:

```
class_name Player
extends Node
var player_health = 2
func _ready():
   print(player_health)
```

Здесь мы видим, что мы называем наш класс **Player**. Теперь мы можем использовать этот тип для указания переменных класса **Player** и даже использовать его для инициирования нового экземпляра класса, например:

```
var player: Player = Player.new()
player.player_health += 1
```

Именование классов — это простой способ типизировать переменные-подсказки с экземплярами наших пользовательских классов.

Производительность

Помимо обнаружения ошибок до их возникновения и наличия автодополнения, подсказки по типу имеют ещё одно большое преимущество в рукаве. Если вы вводите типизированные переменные в своей игре, движку будет гораздо проще с ними работать, что приведёт к повышению производительности.

Поскольку движку не нужно проверять, сможет ли переменная выполнить определённые операции, он может выполнять больше этих операций в секунду. В некоторых случаях это сделает ваш код в два раза быстрее!

Редактор добавляет подсказки по типу

В качестве последнего совета по поводу подсказок по типу, я хотел бы показать вам, что редактор тоже может вам помочь! Если вы зайдёте в **Настройки редактора | Текстовый редактор | Завершение**, там есть настройка под названием **Добавлять подсказки типов**. Эта настройка позволит редактору автоматически дополнять определённые части вашего кода подсказками по типу. Я рекомендую вам включить её:

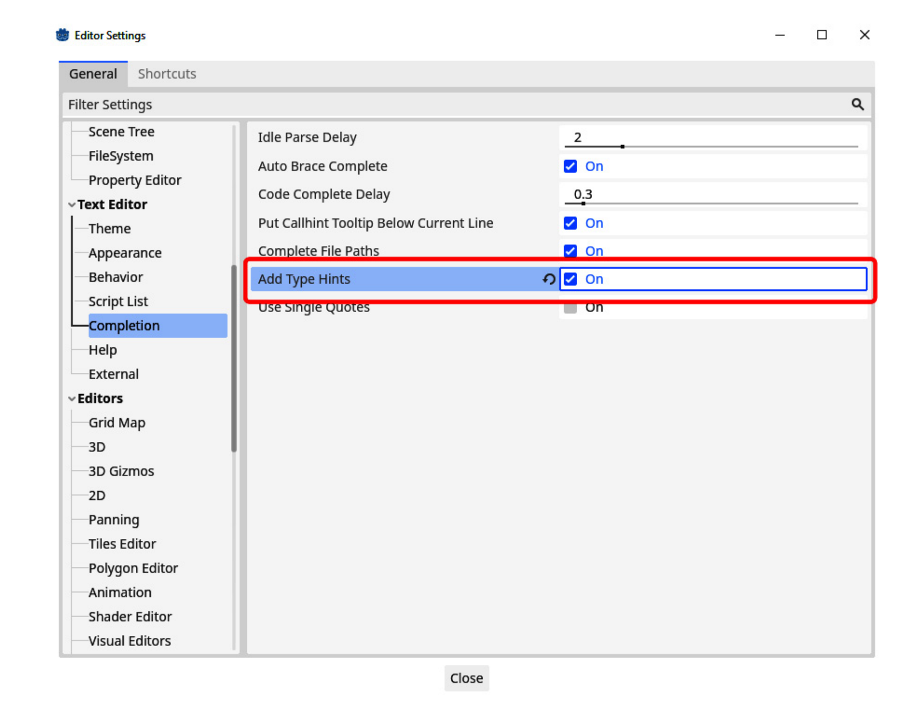


Рисунок 4.3 – Параметр «Добавлять подсказки типов»

Если эта настройка включена, редактор будет автоматически заполнять подсказки по типам всякий раз, когда ему придется генерировать для нас какой-либо код, например, при генерации пустого скрипта.

В этом разделе мы узнали много нового о подсказках типов и увидели, как это может улучшить наш процесс кодирования. Далее давайте рассмотрим очень важную концепцию в программировании: ООП.

Учебник по ООП

Ранее в этой главе мы узнали о функциях, классах и объектах. Эти концепции очень мощные: они дают нам отличный способ работы с данными и логикой, которая их сопровождает.

В программировании существует множество различных

парадигм структурирования кода и данных, одна из них — **объектно-ориентированное программирование** (**ООП**). GDScript — это **объектно-ориентированный** (**ОО**) и **императивный** язык программирования. Это означает, что мы группируем данные и сопутствующую им логику в классы и объекты. Логика, которую мы пишем, состоит из операторов, довольно точно сообщающих компьютеру, что и как делать для нас. Каждый оператор изменяет внутреннее состояние программы. Большинство игровых движков и сопутствующих им языков программирования являются объектно-ориентированными и императивными.

ООП построено на четырех ключевых принципах: наследование (inheritance), абстракция (abstraction), инкапсуляция (encapsulation) и полиморфизм (polymorphism). Итак, давайте рассмотрим их.

Наследование (Inheritance)

ООП позволяет классам наследовать друг от друга. Это означает, что мы получаем всю функциональность родительского класса бесплатно и можем расширить ее дополнительной логикой. Это делает повторное использование кода очень простым.

Например, хотя в игре может быть много разных врагов, большинство из них будут иметь довольно одинаковый код и некоторое количество различающегося кода. Поиск пути, нанесение урона, управление здоровьем, управление инвентарем и т.д. будут общими для почти любого врага. Поэтому мы могли бы определить один класс, **Enemy**, который инкапсулирует все эти функции и от которого все остальные враги могут его наследовать.

Поэтому мы можем определить врагов, которые делают следующее:

1. Приближаются к игроку и используют атаки ближнего боя.
2. Находятся на расстоянии и стреляют в игрока снарядами.

3. Много двигаются и лечат других врагов.
4. И так далее…

Этот список не является исчерпывающим и показывает, что мы можем создать разнообразный состав врагов на основе одного и того же базового класса **Enemy**.

Мы можем наглядно представить это наследование, как и в случае с людьми и их семьями, используя дерево наследования:

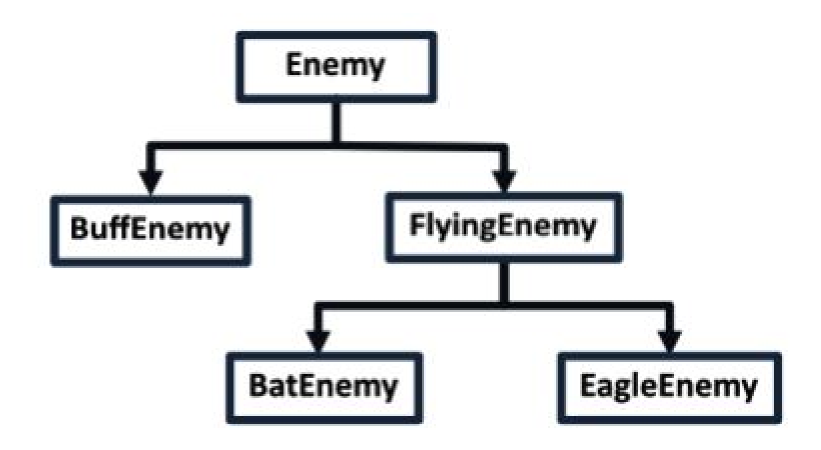


Рисунок 4.4 – Различные виды врагов могут быть легко получены из базового класса Enemy

Рисунок 4.4 наглядно показывает как определённые классы связаны и/или отличаются друг от друга.

Абстракция

Класс скрывает свою внутреннюю реализацию, абстрагируя свою функциональность только путем раскрытия функций более высокого уровня. Пользователю класса всё равно, как достигаются определённые результаты. Насколько известно внешнему миру, фактический процесс получения определённых результатов может быть чистой магией.

Для класса **Enemy** из более раннего примера это может означать, что мы можем попросить врага двигаться к

определённой точке мира, но не определяем как именно. Нам нет дела до того, как враг ищет свой путь или как он перемещается по миру. Это дело врага:

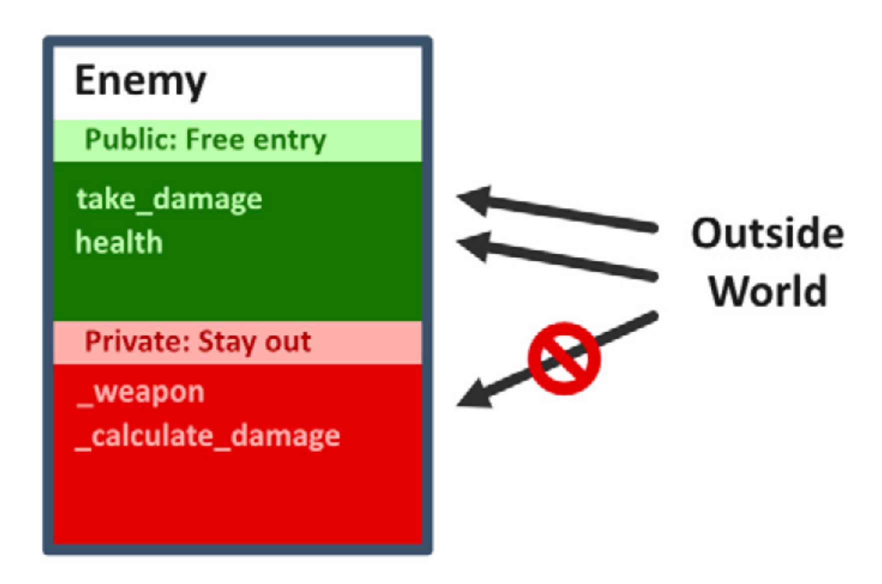


Рисунок 4.5 – Публичные и приватные переменные-члены и методы сообщают внешнему миру, как взаимодействовать с классом

В большинстве языков программирования абстракция представлена в форме публичных и приватных функций-членов и переменных. Они работают следующим образом:

- **Публичные**: внешний мир может получить доступ к переменным и функциям, которые помечены как **публичные (public)** и используются для взаимодействия с объектом.
- **Приватные**: Переменные и функции, отмеченные как **приватные (private)**, недоступны внешнему миру и могут использоваться только самим классом. Они поддерживают внутреннюю функциональность класса.

Однако в GDScript нет способа явно пометить переменные или функции как публичные или приватные. По умолчанию всё публично и доступно внешнему миру. Но есть соглашение, которое разработчики GDScript переняли у разработчиков

Python: мы ставим подчёркивание (_) перед именами переменных и функций, которые должны быть приватными. Таким образом, мы можем сигнализировать, что переменная или метод должны быть приватными и не должны использоваться ничем за пределами класса:

```
extends Node
var health: int = 2 # Публичная переменная
var _weapon: String = "Sword" # Приватная переменная
func take_damage(amount: float): # Публичная функция
   # Получить какой-либо урон
func _calculate_damage() -> float: # Приватная функция
   # Рассчитайте ущерб каким-либо образом
```

Движок не будет принудительно применять такие приватные члены, поэтому вы всё равно можете вызывать их, но это очень плохая практика. Вы можете увидеть это различие между публичными и приватными членами, встроенное в скрипты, которые мы уже написали в функциях, которые уже присутствуют в узлах, таких как функции**_ready()** и **_update()**.

Абстракция имеет множество преимуществ:

- **Безопасность**: поскольку пользователь класса знает, что нужно использовать только публичные методы и переменные, вероятность случайного неправильного использования класса с его стороны снижается.
- **Поддерживаемость**: поскольку функциональность класса скрыта за несколькими публичными функциями, мы можем легко переписать эту функциональность при необходимости, не нарушая другие части кода.

  Это защищает от других классов или фрагментов кода, которые слишком сильно вмешиваются во внутренние части класса. А что, если мы перепишем поиск пути врагов? Если мы правильно инкапсулируем этот код, то проблем не возникнет, но если другие фрагменты кода напрямую вызовут поиск пути врагов, нам придется переписать всё это.

- **Скрытие сложности**: некоторый код может быть очень сложным, но с помощью классов мы можем скрыть это за простыми в использовании методами и переменными-членами.

Теперь, когда мы узнали об абстракции, давайте рассмотрим последний принцип: инкапсуляцию.

Инкапсуляция

Хорошо написанный класс должен инкапсулировать всю важную информацию внутри себя, чтобы пользователю класса не приходилось беспокоиться о мелких деталях. Это означает, что класс должен предоставлять внешнему миру только избранную информацию.

Инкапсуляция — это расширение абстракции, но с привязкой к данным класса. Чем меньше внешнему миру приходится иметь дело с переменными-членами класса напрямую и чем больше — с функциями-членами, тем лучше.

Полиморфизм

Последний принцип ООП — полиморфизм, который гласит, что объекты и методы могут трансформироваться в несколько различных форм. В GDScript это происходит двумя различными способами: через объекты и через методы.

Полиморфизм объектов

Допустим, у нас есть структура класса, как в предыдущем примере: базовый враг, от которого наследуются другие враги. Код может выглядеть примерно так:

```
class Enemy:
   var damage: float
   var health: float
class BuffEnemy extends Enemy:
```

```
   var attack_distance: float = 50
   func _ready():
      damage = 2
      health = 10
class StrongEnemy extends Enemy:
   func _ready():
      damage = 10
      health = 1
```

Теперь, когда мы создаем экземпляры классов **BuffEnemy** и **StrongEnemy**, мы можем указать их как таковые, но мы также можем указать их как их базовый класс **Enemy**:

```
var buff_enemy: BuffEnemy = BuffEnemy.new()
print(buff_enemy.damage)
var enemy: Enemy = buff_enemy
print(enemy.damage)
```

Это работает, потому что всё, что наследуется от класса **Enemy**, должно иметь в своей основе те же переменные-члены и функции, поэтому его можно поместить в переменную родительского класса.

Но вы не можете назначить объект типа **Enemy** переменной, которая типизирована как один из его дочерних классов. Поэтому следующая строка приведёт к ошибке:

```
var buff_enemy: BuffEnemy = Enemy.new()
```

Два дочерних класса также несовместимы. Поэтому следующая строка также выдаст ошибку:

```
var buff_enemy: BuffEnemy = StrongEnemy.new()
```

Эти два предыдущих примера не работают, потому что нет гарантии, что переменные-члены и функции в классах **Enemy** и **StrongEnemy** будут такими же, как в классе **BuffEnemy**. И действительно, мы видим, что класс **BuffEnemy** имеет ещё одну

переменную-член, **attack_distance**, которой нет в классах **Enemy** и **StrongEnemy**.

Хорошей аналогией для концепции полиморфизма являются транспортные средства в реальном мире. Допустим, у нас есть три транспортных средства:

- Автомобили
- Велосипеды
- Грузовики

Хотя все три транспортных средства могут перемещать вас из одной точки в другую, имеют определённое количество колёс и сделаны из металла, существует определённая иерархия:

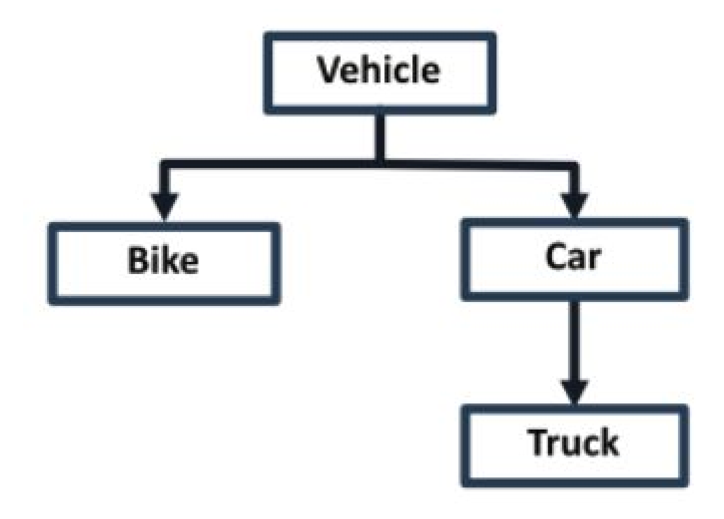


Рисунок 4.6 – Структура классов простых транспортных средств

Велосипеду нужен кто-то, кто будет его приводить в движение, а у автомобиля и грузовика есть моторы. Более того, мы могли бы взять автомобиль в качестве шаблона для грузовика и сказать, что грузовик — это своего рода длинная, большая машина.

Грузовик также отличается от автомобиля тем, что у него есть место для перевозки более крупного груза. Это позволяет нам сказать, что грузовик — это автомобиль, но автомобиль — это

не грузовик.

Переопределение метода

Затем, есть также возможность переопределить методы из родительского класса. Это полностью заменит исходную функцию новой, но только для данного дочернего класса. Это очень полезно, когда дочернему классу нужна некоторая логика, которая немного или даже кардинально отличается от логики родительского класса. Чтобы сделать это в GDScript, метод в дочернем классе должен придерживаться следующих соглашений:

- Имеют одинаковое имя
- Имеют одинаковое количество параметров
- Если параметры типизированы, то имеют одинаковые типы
- Имеют точно такие же значения по умолчанию, если таковые имеются

Вы видите, что нам нужно быть точными, когда мы хотим переопределить метод. Если мы этого не сделаем, движок распознает это как отдельную функцию или ошибку, поскольку переопределение было выполнено неправильно.

Давайте рассмотрим два класса, демонстрирующих это. Базовый класс **Enemy** имеет функцию **die()**, которая выводит "**Aaargh!**". Функция **die** просто выводит восклицание, когда враг умирает. Затем мы наследуем класс **BuffEnemy** от этой базы и переопределяем функцию **die()**, чтобы вывести "**Как ты победил меня?!?**":

```
class Enemy:
   func die():
      print("Aaargh!")
class BuffEnemy extends Enemy:
   func die():
      print("Как ты победил меня?!?")
```

Если вы вызовете функцию **die()** каждого типа врага, вы

увидите, что у каждого из них есть своя собственная реализация функции:

```
var enemy: Enemy = Enemy.new()
enemy.die()
```

Предыдущий код выведет "**Aaargh!**", , как и ожидалось. Теперь вызовем её для класса **BuffEnemy**:

```
var buff_enemy: BuffEnemy = BuffEnemy.new()
buff_enemy.die()
```

Теперь мы выполняем переопределённую функцию **die()**, и в выводе будет написано: "**Как ты победил меня?!?**".

Даже если вы поместите объект **BuffEnemy** в переменную **Enemy**, он всё равно будет использовать переопределённую функцию из класса **BuffEnemy**:

```
var enemy: Enemy = BuffEnemy.new()
enemy.die()
```

Мы опять увидим вывод "**Как ты победил меня?!?**". Это происходит потому, что класс **BuffEnemy** наследуется от **Enemy** и, таким образом, имеет тип **Enemy**, но реализация его функций всё ещё может быть переопределена.

Мы много узнали об ООП и его принципах. Это очень интересная, но сложная тема. Не беспокойтесь слишком сильно о том, чтобы сразу освоить в совершенстве все принципы. Знание того, что они существуют, уже половина работы. Давайте завершим главу несколькими дополнительными упражнениями.

Дополнительные упражнения – Заточка топора

1. Напишите функцию **limit_inventory()**, которая принимает массив, представляющий инвентарь, и целое число. Функция проверяет, длиннее ли массив, чем предоставленное целое число; если да, она должна удалить лишние элементы. Наконец, функция возвращает результирующий массив:

```
var inventory: Array = ["Сапоги", "Меч", "Виноград"
var limited_inventory: Array = limit_inventory(inve
print(limited_inventory)
```

   В этом примере должно быть выведено **["Boots", "Sword"]**.

2. Перепишите предыдущую функцию так, чтобы принимаемое ею целое число имело значение по умолчанию 3, чтобы следующий код работал:

```
var inventory: Array = ["Boots", "Sword", "Grapes",
var limited_inventory: Array = limit_inventory(inve
print(limited_inventory)
```

   Это должно вывести **["Boots", "Sword", "Grapes"]**.

3. Перепишите этот код так, чтобы он больше не вызывал ошибок:

```
func _ready():
   var player_health = 5
   if player_health > 2:
      var damage = 2
   player_health -= damage
```

4. Напишите классы **Player** и **Enemy**, которые заставят следующий код работать. В этом коде игрок и враг будут наносить друг другу урон до тех пор, пока здоровье одного из них не станет равным или меньше нуля. Рассматривайте это как примитивную битву::

```
var player: Player = Player.new()
var enemy: Enemy = Enemy.new()
while player.health > 0 and enemy.health > 0:
   enemy.take_damage(player.damage)
   player.take_damage(enemy.damage)
```

5. Перепишите классы **Player** и **Enemy** из предыдущего упражнения так, чтобы они наследовались от того же базового класса..

Итоги

С функциями, классами и подсказками типов в нашем инструментарии мы наконец-то изучили все основные строительные блоки программирования! Отныне ваши возможности безграничны!

В следующей главе мы узнаем, как писать и структурировать наш код так, чтобы его было легко использовать и понимать другим.

Опрос

- Почему мы используем функции и классы?
- Для каких двух целей можно использовать ключевое слово **return** в функциях?
- Какова область видимости переменной? Каковы различные уровни видимости?
- Какова область действия функции?
- Является ли класс группой переменных и функций?
- Как создать новый экземпляр класса **Enemy**, используя следующий код?

```
class Enemy:
   var damage: int = 5
   # Остальная часть класса
```

```
var new_enemy: Enemy = ...
```

- Как вызвать экземпляр класса?
- Что такое подсказка типа?
- Добавьте подсказку типа к следующим переменным:
    - **var player_health = 5**
    - **var can_take_damage = true**
    - **var sword = { "damage_type": "fire", "damage": 6 }**
- Какое ещё преимущество использования подсказок типов, помимо автодополнения и повышения производительности?

5

Как и почему нужно поддерживать чистоту кода

В *Главах 1–4* мы изучили все основы программирования и собираемся погрузиться в разработку нашей собственной игры.

Но прежде чем мы это сделаем, мы должны осознать, что кодовая база для игр может стать очень большой. Это означает, что код и системы, которые мы пишем однажды, могут быть погребены под другим кодом и системами. В результате возвращение к нашей предыдущей работе может быть хлопотным, потому что мы забываем, как или почему мы кодировали определённые вещи определённым образом.

Вот почему сейчас идеальный момент остановиться и подумать о том, как сохранить наш код чистым и понятным даже спустя месяцы после его написания. Большинство вещей в этой главе были изучены мной на собственных ошибках и поиске решения в книгах и статьях.

Хотя большинство советов могут показаться проявлением критического мышления и приведут вас к той же мысли (что, скорее всего, и произойдет), всегда полезно озвучить их и объяснить, почему программисты их используют.

Помня о них во время программирования, вы значительно превзойдёте других начинающих программистов.

В этой главе мы рассмотрим следующие основные темы:

- Именование вещей (снова)
- Написание хороших функций
- Зачем использовать приватные переменные и функции
- Не повторяйтесь (DRY)
- Защищённое программирование

- Руководства по стилю кодирования

Технические требования

Если вы где-то застрянете, проверьте папку **chapter05** в репозитории примеров кода. Репозиторий можно найти здесь: https://github.com/PacktPublishing/Learning-GDScript-by-Developing-a-Game-with-Godot-4/tree/main/chapter05.

Возвращаемся к именованию вещей

Давайте еще раз рассмотрим, как называть переменные, функции и классы. Выбор правильного имени для любого из них очень важен, поскольку это значительно облегчит понимание кода.

Соглашения об именовании

Мы видели в *Главах 3* и *4*, что имена переменных, функций и классов имеют различные ограничения. Мы использовали определённые правила для именования каждого из них. Эти способы называются **соглашениями об именовании (naming conventions)**. Они содержат несколько терминов, описывающих ограничения для формирования имён. Три основных соглашения об именовании, которые рекомендуются в руководстве по стилю GDScript:

- **snake_case**: Мы использовали это соглашение об именовании для именования всех наших переменных и методов. Оно называется **snake_case**, потому что все слова в имени соединены подчёркиваниями, которые выглядят как маленькие змеи. Примерами имён могут быть следующие: **player_health**, **movement_speed** и **weekly_highscore**.
- **SCREAMING_SNAKE_CASE**: Это соглашение, которое мы

используем для именования констант. Оно точно такое же, как и в случае snake_case, только все алфавитные символы заглавные. Примеры имён:**BUTTON_SIZE**, **PI** и **TEAM_A_COLOR**.
- **PascalCase**: Мы использовали это соглашение об именовании для именования классов и узлов в сцене. Оно называется **PascalCase**, потому что было популяризировано в языке программирования **Pascal**. В этом соглашении мы начинаем каждое слово в имени с заглавной буквы, а все остальные — с маленькой. Примеры имён: **BackgroundColor**, **PlayerWeapon** и **GameStartTimer**.

Есть множество других, более экзотических соглашений, например: **kebab-case**, **camelCase**, **flatcase** и т.д. Но они не используются в GDScript.

Общие советы по именованию

Теперь позвольте мне прояснить: давать имена вещам нелегко. На самом деле, это одна из самых сложных вещей в программировании. Так что, если вы правильно придумываете названия, вы всегда сможете быстро вернуться к любому коду.

Вот несколько советов, которые помогут вам стать мастером имён.

Используйте осмысленные и описательные имена

В прежние времена программистам приходилось работать с компьютерами, у которых не было большой вычислительной мощности и памяти, и они гордились тем, что создавали самый короткий скрипт для решения проблемы. Это приводило к коду, в котором переменным давали одно- или двухбуквенные имена, такие как **a** или **c5**. Оптимизация фрагмента кода с целью сделать его максимально коротким — это очень приятно. Однако такие фрагменты кода очень непонятны. Даже тот, кто написал скрипт, может прочесть не его по прошествии некоторого времени.

Некоторые даже заходят так далеко, что говорят, что короткие имена переменных приводят к более производительной игре. Это совсем не так. Язык программирования будет токенизировать переменные в коде, заставляя любое имя переменной работать одинаково быстро, независимо от его длины.

Вот почему описательные имена переменных — это большой плюс. Конечно, их ввод занимает несколько дополнительных секунд, но это ничто по сравнению с минутами или даже часами, потраченными на выяснение того, почему переменная существует и как её следует использовать. Кроме того, автодополнение всегда нам поможет.

Хитрость заключается в том, чтобы сделать имя переменной, метода или класса осмысленным и описательным. Чтобы сделать это, вы можете задать себе следующие вопросы для разных типов:

- **Переменные**:
  - Какие данные будет содержать переменная?
  - Как следует использовать эти данные?
- **Функции**:
  - Что делает эта функция?
  - Какие данные возвращает функция?
  - Какие параметры требуются функции для работы?
- **Классы**:
  - Для чего будет использоваться класс?
  - За какие данные отвечает класс?

Использование этих вопросов поможет вам принять решение при выборе названий для переменных, функций и классов.

Избегайте слов-паразитов

Хотя длинные описательные названия — это правильный путь,

мы также не хотим перегружать название лишними или ненужными словами, такими как:

- The
- A
- Object

Такие слова просто загромождают имя и делают его неоправданно длинным, не привнося при этом никакого дополнительного смысла.

Делайте имена легкопроизносимыми

Хороший код должен быть легко читаемым. Это значит, что вы должны иметь возможность читать его как книгу, и он должен иметь смысл без необходимости смотреть на содержимое функции или тип данных переменных.

Это также означает, что вы должны сохранять имена произносимыми. Используйте полные слова или очень распространенные сокращения, если вы решили сократить что-нибудь.

Будьте последовательны

Теперь самый важный совет: будьте последовательны в именовании. Таким образом, вы сможете рассчитывать на свой собственный стиль именования и делать предположения о переменных, функциях и классах, которые вы пишете. Если вы нарушаете какие-либо правила, по крайней мере, нарушайте их последовательно, а не просто делайте что-то другое каждый раз.

Публичные и приватные члены класса

В *Главе 4* мы узнали, что **абстракция** и **инкапсуляция** являются двумя ключевыми компонентами **объектно-ориентированного программирования**. Это означает, что код вне класса не должен беспокоиться о том, как этот класс получает результаты. Для всего внешнего мира это может быть

магия или, что ещё хуже, ручной труд.

Чтобы обозначить, что конкретная переменная или метод предназначены только для внутреннего использования классом, GDScript перенял соглашение, пришедшее из Python: подчёркивание перед именем этой переменной или функции.

```
class Enemy:
	var damage: float = 100.0
	var _damage_multiplier: float = 2.0


func _ready():
	var enemy: Enemy = Enemy.new()
	enemy.dam
	        .P damage
	        .P _damage_multiplier
```

Рисунок 5.1 – Автодополнение по-прежнему предлагает приватные переменные

Однако, как показано на *Рисунке 5.1*, вы заметите, что автодополнение всё ещё предлагает приватные члены класса. Всё ещё очень важно указать, какие члены класса являются приватными и к которым не следует обращаться. Это поможет вам или любому другому программисту, который придет после вас, использовать этот класс.

Создавайте короткие функции

Чем больше функция пытается сделать, тем больше кода в этой функции и тем сложнее понять, что она делает. Поэтому, чтобы функции были легко понятными, хорошим правилом является ограничение числа строк до 20. Это позволяет быстро понять, что происходит и как эффективно использовать функцию.

Конечно, вы можете вызывать разные функции. Разделение длинных функций на несколько меньших с хорошим

описательным именем сэкономит вам много часов на выяснение того, что делает этот код.

DRY — don’t repeat yourself / не повторяйтесь

Есть две аббревиатуры, о которых слышал почти каждый студент-программист. Первая — DRY. Эта аббревиатура призывает нас писать фрагмент кода только один раз, а затем использовать его как можно чаще. Если мы создадим небольшие, общие функции, мы сможем предотвратить копирование и вставку одних и тех же строк по всей нашей кодовой базе.

Но я также должен предупредить вас, чтобы вы не переусердствовали. Иногда лучше иметь немного дублирующего кода, который лучше подходит для конкретного скрипта, чем впихивать несколько скриптов в один кусок кода. Пользуйтесь здравым смыслом.

KISS — keep it simple, stupid / делайте вещи проще

Вторая аббревиатура, которую все знают, это **KISS** , что означает **keep it simple, stupid**. Это можно интерпретировать двумя способами:

- Сведите решение к минимуму, что означает, что вы не решаете проблемы, которых ещё не существует. Таким образом, вы не разрабатываете функции, которые не нужны, и не тратите время на создание того, чем никто не будет пользоваться.
- Не усложняйте свой код. Сложный код, как известно, трудно поддерживать и понимать. Вот почему лучше сохранять любое решение простым, чтобы вы всегда знали, что происходит.

Простой код всегда легче читать, понимать и поддерживать. Так что делайте вещи проще!

Защищёное программирование

Последний принцип, который я хочу вам показать, — это **защищёное программирование**. В этой парадигме вы пытаетесь делать всё наверняка, проверяя как можно больше вещей и пограничных случаев в коде. Например, для функции вы можете проверить в начале функции, являются ли параметры правильными. Таким образом, вы предотвратите множество сбоев в долгосрочной перспективе.

Положим, у вас есть функция, которая должна возвращать элемент из инвентаря по определенному индексу, вы можете написать её без защиты и с защитой:

```
func get_inventory_item(index: int):
   return inventory[index]
func  get_inventory_item (index: int):
   if index < 0 or index >= inventory.size():
      return
   return inventory[index]
```

Вторая версия функции является защитной, поскольку она сначала проверяет, находится ли индекс нужного нам предмета в пределах диапазона инвентаря. Мы делаем это, потому что если индекс находится вне этого диапазона (range), мы получаем падение игры.

Руководства по стилю программирования

Наконец, я хотел бы рассказать, что такое руководства по

стилю программирования. Это руководства, которые рассказывают вам, как структурировать ваш код. Эти руководства никогда ничего не говорят о содержании кода, а больше о том, как его стилизовать.

Вы можете сравнить рекомендации этих руководств со стилями для книг. Я мог бы поместить все предложения в одну длинную строку без стилей, заголовков или изображений. Но в конечном итоге это сделало бы содержание очень сложным для понимания.

Помимо того, что эти руководства по стилю делают код более читабельным, они также позволяют целым командам кодеров работать на одной волне, потому что код каждого человека выглядит похоже, и людям не приходится постоянно переключаться между разными стилями кодирования, пытаясь понять кодовую базу проекта.

У большинства компаний есть свое внутреннее руководство по стилю. Клонечно же есть официальное руководство по стилю GDScript! Вы можете прочитать его здесь:https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_styleguide.html.

Я не рекомендую вам читать всё это и пытаться применить всё это сразу. Вместо этого вы можете прочитать некоторые фрагменты здесь и там, и как только вы усвоите эти рекомендации, прочтите ещё немного и попытайтесь подогнать их под свой собственный стиль кодирования.

Не заблуждайтесь. Даже с этими руководствами по стилю всё ещё есть место для личного подхода при кодировании. Эти руководства будут просто рамками, которые сделают ваш код более приятным и понятным для других программистов, работающих с тем же языком программирования и фреймворком.

В оставшейся части этой главы я хотел бы рассмотреть некоторые советы из официального руководства по стилю GDScript. А именно:

- Пробелы
- Пустые строки
- Длина строк

Итак, давайте перейдем непосредственно к рекомендациям по стилю от разработчиков Godot Engine.

Пробелы

За исключением отступов, которые мы обсуждали в *Главе 2*, GDScript не заботится о пробелах внутри строк кода. Следующие две строки функционально одинаковы:

```
var total_damage:float=100+get_damage()*0.5
var total_damage: float = 100 + get_damage() * 0.5
```

Однако вторая строка гораздо более читабельна для людей, потому что каждая часть имеет пространство визуально отделяющее её от других. Вот почему важно всегда использовать пробел между числами, вызовами функций и операторами. Строка не будет просто мешаниной символов.

В массивах мы также добавляем пробелы между элементами, чтобы чётко показать, что каждый из них является отдельной сущностью:

```
inventory = ["Boots","Sword","Potion"]
inventory = ["Boots", "Sword", "Potion"]
```

Обратное тоже может быть верно. Вставка ненужных пробелов может скрыть некоторые операторы, например, доступ к ключу в словаре:

```
dictionary ["key"] = 100
dictionary["key"] = 100
```

В предыдущем примере более очевидно, что квадратные скобки используются для доступа к ключу из словаря, а не для

определения массива.

Два других примера, где отсутствие пробела показывает чёткую связь между элементами, — это имя функции и список её параметров:

```
print ("Hello")
print("Hello")
```

Это также можно увидеть при доступе к переменной-члену или функции из объекта:

```
object . function()
object.function()
```

В общем, очень важно использовать пробелы в строке кода, чтобы показать, когда вещи являются отдельными сущностями и не использовать их, когда эти вещи принадлежат друг другу. Это значительно улучшит читаемость.

Пустые строки

Другой тип пробелов, который мы можем использовать, чтобы сделать код более читабельным, — это пустые строки. Пустая строка — это просто строка, которая ничего не содержит. Руководство по стилю предлагает использовать две пустые строки для разделения функций и определений классов. Таким образом, становится ясно, какие фрагменты текста относятся друг к другу как функция:

```
func deal_damage(amount: float) -> void:
   player_health -= amount
func heal(amount: float) -> void:
   player_health += amount
```

В дополнение к разделению функций и классов двумя пустыми строками, руководство советует нам использовать одну пустую строку для разделения строк кода, которые логически

сгруппированы. Например, если у нас есть код, который вычисляет урон от атаки, а затем применяет этот урон ко всем врагам, мы можем красиво сгруппировать эту логику следующим образом:

- Определение ущерба
- Расчёт общего ущерба
- Нанесение урона всем врагам

Мы можем увидеть это в следующем коде:

```
var weapon_damage: float = 10
var damage_type: String = "Огонь"
var total_damage: float = weapon_damage
if damage_type == "Электричество":
   total_damage *= 2
for enemy in enemies:
   enemy.deal_damage(total_damage)
```

Вы увидите, как я грешу против правил руководства пустыми строками тут и там по всей книге. Я делаю это в основном для того, чтобы сделать код более компактным и уместиться на страницах, но это не делает правило менее важным!

Длина строки

Раньше компьютерные мониторы были крошечными. Они часто не могли вместить более 70–90 символов в одной строке текста, прежде чем он прокручивался за пределы или переносился. Вот почему код лучше всего писать строками, не превышающими эту длину. Сейчас мой сверхширокий компьютерный монитор может вместить более 500 символов в одной строке без проблем. Ну, в любом случае, это не техническая проблема. Работа с текстом такой ширины делает его очень сложным для чтения людьми!

Вот почему люди все еще ограничивают длину строк при программировании, чтобы все было красиво и легко читалось. Хотя, конечно, не все согласны с идеальной длиной строки,

GDScript по умолчанию устанавливает 80 символов в качестве мягкого ограничения и **100** в качестве жёсткого ограничения.

Рекомендуется не превышать это количество символов в строках. Если вы столкнетесь с ними, вы всегда можете разделить строку, сохранив промежуточные результаты в отдельных переменных. Например, следующий фрагмент кода проверяет, находится ли здоровье игрока в диапазоне от **0** и до **100** и есть ли у игрока зелье в инвентаре:

```
if player_health > 0 and player_health < 100 and invento
   # Take potion
```

Технически это не слишком долго, но чтобы показать, как можно уменьшить длину строки и даже сделать условие в операторе **if** более читабельным, давайте перепишем этот фрагмент следующим образом:

```
var can_heal: bool = player_health > 0 and player_health
var has_potion: bool = inventory.has("Зелье")
if can_heal and has_potion:
   # Принять зелье
```

Как видите, теперь нет слишком длинных строк, а оператор **if** стал гораздо более читабельным.

Использование документации

Иногда трудно предугадать, как работают встроенные классы, функции или переменные Godot Engine. К счастью, у движка есть отличная документация, которая объясняет всё очень подробно.

Доступ к документации класса

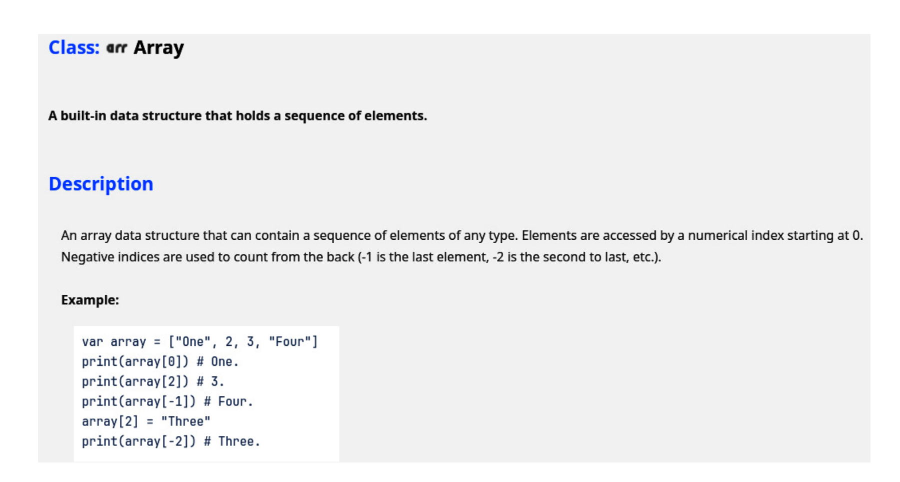
Мы всегда можем получить доступ к документации любого внутреннего класса, просто используя *Ctrl* + щелчок по имени

класса. Это перенесёт вас на конкретную страницу документации этого класса.

```
func _ready():
    var items: Array = []
```

Рисунок 5.2 – Ctrl + щелчок по имени внутреннего класса, например, класса Array

Как показано на *Рисунке 5.3*, страница документации начинается с простого описания того, для чего используется класс, а иногда в этой части приводятся примеры использования.



Class: **Array**

A built-in data structure that holds a sequence of elements.

Description

An array data structure that can contain a sequence of elements of any type. Elements are accessed by a numerical index starting at 0. Negative indices are used to count from the back (-1 is the last element, -2 is the second to last, etc.).

Example:

```
var array = ["One", 2, 3, "Four"]
print(array[0]) # One.
print(array[2]) # 3.
print(array[-1]) # Four.
array[2] = "Three"
print(array[-2]) # Three.
```

Рисунок 5.3 – Страница документации для класса Array

Затем следует обзор функций-членов класса, переменных, сигналов и операторов. Мы увидим больше о сигналах в *Главе 9*. Обратите внимание, что вы можете легко щелкнуть по именам функций или переменных, чтобы напрямую перейти к их объяснению.

Важное примечание

Помните, что функции также можно называть методами, а переменные — свойствами. Это потому, что функции,

привязанные к классу, называются методами, а переменные — свойствами класса.

После раздела обзора следует подробное описание каждой функции и переменной. Для функций мы получаем объяснение того, что делает функция, какие параметры она принимает и какой тип данных возвращает.

- bool is_empty() const

  Returns true if the array is empty.

Рисунок 5.4 – Раздел документации с примером функции

Для переменных мы также получаем описание того, для чего используется эта переменная и какой тип данных должен иметь её значение.

- Vector2 position [default: Vector2(0, 0)]
  set_position(value) setter
  get_position() getter

  Position, relative to the node's parent.

Рисунок 5.5 – Раздел документации с примером переменной

Такой способ доступа к документации очень удобен, если мы хотим получить общее представление о том, что делает класс.

Прямой доступ к документации функции или переменной

Чтобы напрямую перейти к документации функции или переменной, вам просто нужно нажать *Ctrl* + щелкнуть по этой функции или переменной, и вы перейдете непосредственно к

соответствующему разделу.

```
func _ready():
	var items: Array = []
	items.is_empty()
	
```

Рисунок 5.6 – Удерживая клавишу Ctrl и нажимая на функцию или переменную, вы сразу перейдете в нужный раздел документации

После нажатия на ссылку мы сразу попадаем в раздел с *Главы 5.4*.

Переходим к определению функции или переменной

Это сочетание клавиш также работает с нашим собственным кодом: если удерживать *Ctrl* + щелчок в Windows или Linux или *Option* + щелчок в Mac на функции или переменной, которые мы где-то определили, редактор покажет, где эта функция или переменная была определена в коде.

В качестве эксперимента попробуйте использовать это сокращение для различных функций, которые мы определили для классов **Enemy**, созданных в *Главе 4*.

Поиск документации

Вы также можете искать все классы, функции и переменные. Просто выполните следующие шаги:

1. Нажмите *F1*, и появится строка поиска.
2. Введите любой класс, функцию или переменную, которую вы хотите найти.
3. Выберите правильный результат поиска.

Результат показан на *Рисунке 5.6*:

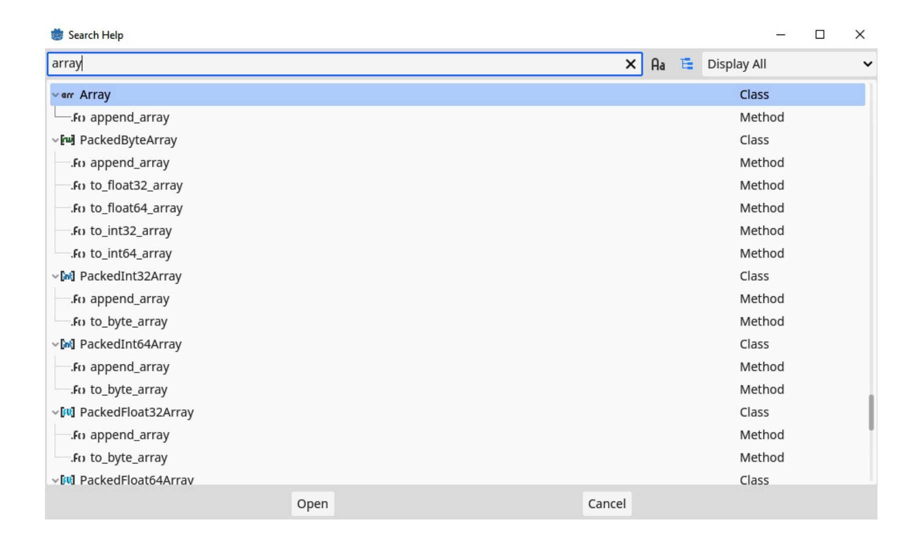


Рисунок 5.7 – Вы также можете выполнить поиск по всей документации, нажав клавишу F11 на клавиатуре

Это упрощает поиск нужного раздела в документации, если вы не можете удерживать *Ctrl* + щелчок в Windows и Linux или нажать *Option* + щелчок в Mac на классе, функции или переменной в вашем коде.

Доступ к онлайн-документации

Вся эта документация также размещена в сети. В сетевой версии есть некоторые страницы и руководства, к которым вы не можете получить доступ в офлайн-версии. Также в сетевой версии проще открывать несколько страниц документации.

Просто перейдите по ссылке: https://docs.godotengine.org/.

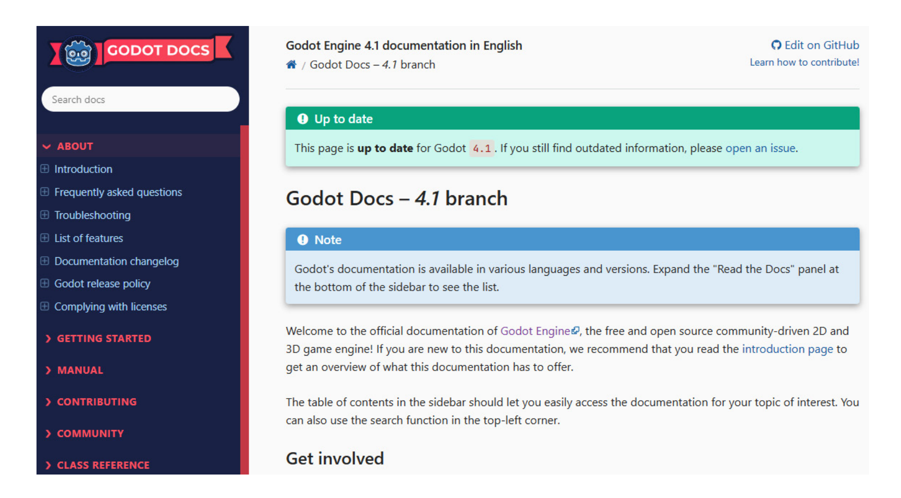


Рисунок 5.8 – Онлайн-документация

В строке меню слева вы можете просмотреть все доступные статьи и перейти к разделам, которые вас интересуют.

Краткое содержание

Вот и всё! В *Части 1* книги мы учились программировать, а в этой главе мы дали несколько дополнительных советов о том, как стать хорошим программистом. Помните, ни один совет в этой главе не высечен на камне и не навязывается движком. Поэтому вы можете нарушать их при необходимости. Но они здесь для вашего же блага, и многие программисты приняли их в качестве ежедневной практики.

В следующей главе, которая также находится в начале следующей части книги, мы наконец начнем работать над нашей игрой! Надеюсь, вы так же взволнованы, как и я!

Опрос

- Какие три соглашения об именовании используются в Godot Engine и GDScript?
- Правильно ли названы следующие функции?

- ○ **CalculateLifePoints()**
- ○ **stop_moving()**
- ○ **do_a_thing()**
- ○ **drawcircles()**

- Правильно ли названы следующие классы?

  - ○ **Player**
  - ○ **normal_enemy**
  - ○ **MOTORCYCLE**

- Что означают аббревиатуры DRY и KISS?

Часть 2: Создание игры в Godot Engine

Освоив основы программирования, мы наконец-то начнём работать над собственной игрой с нуля. В этой части мы узнаем всё о гибкой системе узлов Godot Engine и создадим игру в стиле *Vampire Survivors*.

К концу этой части вы создадите целую игру, используя различные узлы и методы разработки игр. Вы даже сможете играть в игру со своими друзьями, потому что мы закончим эту часть главой о том, как сделать игру многопользовательской.

Эта часть состоит из следующих глав:

- *Глава 6*, *Создание собственного мира в Godot*
- *Глава 7*, *Заставляем персонажа двигаться*
- *Глава 8*, *Разделение и повторное использование сцен*
- *Глава 9*, *Камеры, столкновения и предметы коллекционирования*
- *Глава 10*, *Создание меню, создание врагов и использование автозагрузок*
- *Глава 11*, *Совместная игра в многопользовательском режиме*

6

Создание собственного мира в Godot

В *Части 1* этой книги вы изучили основы программирования! Немалый подвиг, если вы спросите меня. Так что поздравляю с этим достижением! Теперь пришло время связать все это вместе и начать работать над нашей игрой.

На заре разработки игр все происходило через код. Компьютерный волшебник должен был программировать все, от систем и функций до уровней и размещения ресурсов. В последнее время инструментарий для создания игр стал намного лучше, стал бесплатным и очень удобным для пользователя.

Godot, как и большинство современных игровых движков (Unity, Unreal Engine, Construct и другие), имеет графический интерфейс, который позволяет легко перетаскивать элементы нашей игры на уровни или другие сцены. В этой главе мы научимся использовать этот графический интерфейс, создав элементарного персонажа игрока и небольшой мир для его обитания.

Мы также изучим несколько приемов, позволяющих связать код и графический редактор вместе с помощью ссылок на узлы и экспорта переменных.

В этой главе мы рассмотрим следующие основные темы:

- Узловая система Godot
- Создание игрового персонажа
- Ссылки на узлы в скриптах
- Экспорт переменных
- Создание элементарных форм

Технические требования

Поскольку мы создадим игру с нуля, я взял на себя смелость предоставить вам основу проекта. Вы можете найти этот базовый проект в папке для этой главы в подпапке **/start**. Этот проект предоставляет некоторые ресурсы, такие как изображения и звуки. Создание этих ресурсов выходит за рамки этой книги. Полученные файлы проекта для этой главы можно найти в **/result** папки этой главы.

В последующих главах вы найдете итоговый проект в **root** папке этой главы. Предполагается, что вы используете результаты из предыдущей главы в качестве отправной точки.

Итак, скачайте начальный проект и приступим: https://github.com/PacktPublishing/Learning-GDScript-by-Developing-a-Game-with-Godot-4/tree/main/chapter06/start.

Игровой дизайн

Прежде чем бездумно создавать игру, давайте спланируем, какую игру мы хотим сделать. Это структурирует наши мысли и гарантирует, что мы будем работать над игрой, которую хотим сделать, не делая ненужных обходных путей. Лучший способ сделать это — через **дизайн-документ игры** (**GDD**). Хотя для такого рода документа нет установленного формата, он должен в конечном итоге ответить на некоторые основные вопросы об игре:

- К какому жанру относится игра?
- Какие механики будут в игре?
- Что это за история?

Некоторые документы по дизайну игр занимают сотни страниц. Но поскольку это не книга по дизайну игр, давайте определим нашу игру относительно этих трёх вопросов, а затем разберём детали по ходу дела.

Жанр

Недавно мы стали свидетелями рождения нового жанра, известного как **вампирские игры-выживалки**, также известные как игры **VS**. В этом типе игр вы управляете персонажем в 2D-мире сверху вниз. Персонаж должен победить волны монстров, преследующих его, стреляя в них. Игрок может управлять персонажем, перемещая его, но стрельба происходит автоматически. Она не требует ввода.

Этот жанр имеет огромную базу игроков, а базовая игра относительно проста в реализации и в то же время интересна для игры. Поэтому это был бы идеальный тип игры для воссоздания в следующих главах.

Механики

В жанре выживалок есть несколько основных механик, которые очень важно реализовать правильно:

- **2D — мир**: игровое поле представляет собой 2D плоскость, на которой у нас есть вид сверху. Некоторые из них действительно реализованы в 3D, но механика все ещё работает в 2D.
- **Движение персонажа**: нам необходимо иметь возможность перемещать персонажа по миру.
- **Волны врагов**: нам нужны враги, которые угрожают убить игрока, и нам нужно создавать их так, чтобы они представляли собой настоящую проблему.
- **Автоматическая стрельба**: персонаж игрока будет автоматически стрелять снарядами, целясь во врагов.

Теперь, когда мы определились с жанром и основными механиками, давайте проработаем историю, на которой будет основана наша игра.

История

Давайте не будем слишком обременять себя написанием всей

истории. В играх история может быть рассказана также через то, как выглядит и ощущается игра. Поэтому мы можем указать общую обстановку, которая связывает весь опыт воедино.

Как насчет такого сеттинга: Вы средневековый рыцарь, сражающийся на королевском турнире, чтобы найти сильнейшего солдата во всей стране. Вам придется сражаться с несколькими врагами, такими как орки и тролли, в нескольких раундах, каждый из которых сложнее предыдущего. Единственное оружие, которое вам дано, это лук, с помощью которого вы можете стрелять стрелами в своих противников.

Теперь, когда у нас есть представление о том, какую игру мы создаем, давайте приступимк её созданию!

Создание игрового персонажа

Начнём с создания простейшего персонажа для нашей игры:

1. Откройте файл **main.tscn**, который я предоставил в базе проекта.
2. Выберите корневой узел (**root**), названный **Main**, и нажмите кнопку **Add Child Node**:

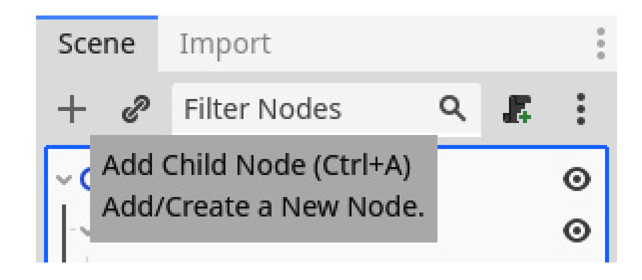


Рисунок 6.1 – Кнопка добавления нового дочернего узла к выбранному узлу в дереве

1. Затем найдите и добавьте узел **Node2D**. Вы можете использовать строку поиска вверху, чтобы облегчить поиск узла. Это узел, имеющий позицию в 2D-пространстве:

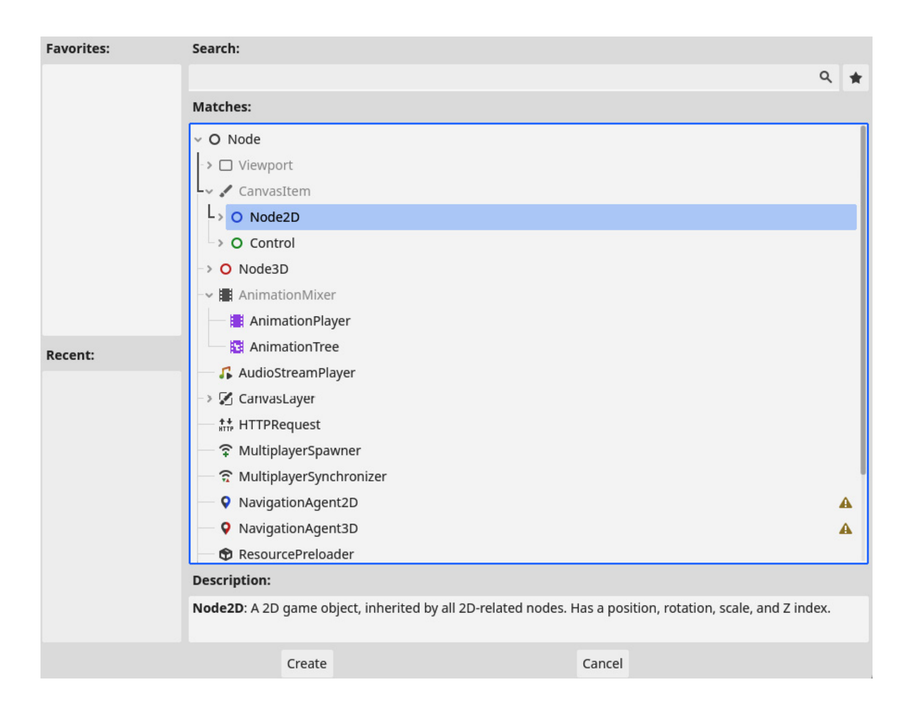


Рисунок 6.2 – Поиск и выбор узла Node2D

1. Затем переименуйте этот Node2D в **Player**, щёлкнув правой кнопкой мыши по узлу и выбрав **Переименовать**, как мы это делали в *Главе 2*.

Player будет базовым узлом для нашего персонажа игрока. Сюда мы добавим все остальные узлы, составляющие сцену **Player**. Первым из этих узлов будет спрайт.

Добавление спрайта

Первое, что мы можем сделать, чтобы оживить нашего персонажа игрока, — это придать ему визуальный образ, что-то, с чем игрок может соотнести себя как с главным героем. Снова выполните *Шаги 1 — 3* в разделе *Создание персонажа игрока*, чтобы добавить узел **Sprite2D** к узлу **Player**, чтобы дерево сцены выглядело следующим образом:

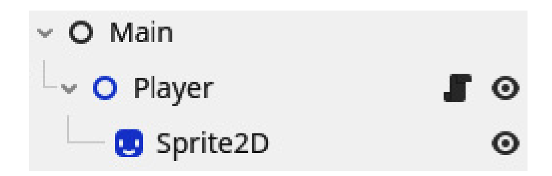


Рисунок 6.3 – Дерево сцены на данный момент

Узел **Sprite2D —** это узел, который может отображать изображение, также называемое **спрайтом**. Если вы нажмёте на **Sprite2D**, вы увидите, что панель **Инспектор** справа отобразит информацию об этом узле:

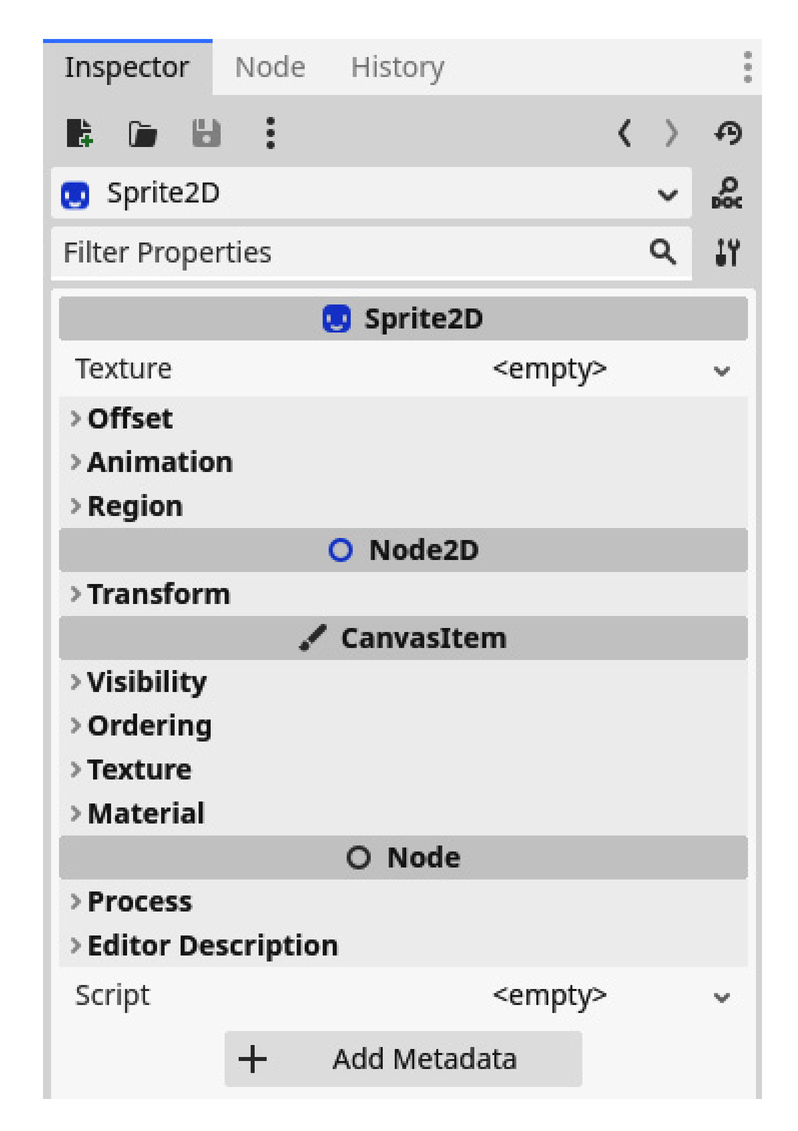


Рисунок 6.4 – Инспектор отображает узел Sprite2D

Есть настройки для свойств **Texture**, **Offset**, **Animation**, **Region** и т.д. Вы можете просмотреть их, чтобы получить представление обо всех доступных настройках. Различные вкладки называются **Группы свойств (Property Groups)**, а сами настройки называются **Properties**.

Нас интересует только свойство **Texture**, поскольку именно

здесь мы можем задать изображение, которое будет отображать этот узел. Итак, давайте добавим спрайт для нашего персонажа!

1. В области **Файловый менеджер (FileManager)** перейдите к **assets/sprites/character**.

   Здесь вы найдёте множество готовых спрайтов персонажей.

Kenney assets

Все ресурсы, которые мы используем в этой книге, принадлежат Кенни и могут быть использованы в любом проекте. Вы можете найти больше его замечательных ресурсов на https://kenney.nl/.

1. Перетащите любой из них на свойство **Texture** узла **Sprite2D**. Я использую текстуру **character01.png**.
2. Панель **Инспектор** теперь выглядит **Sprite2D** теперь должен выглядеть примерно так:

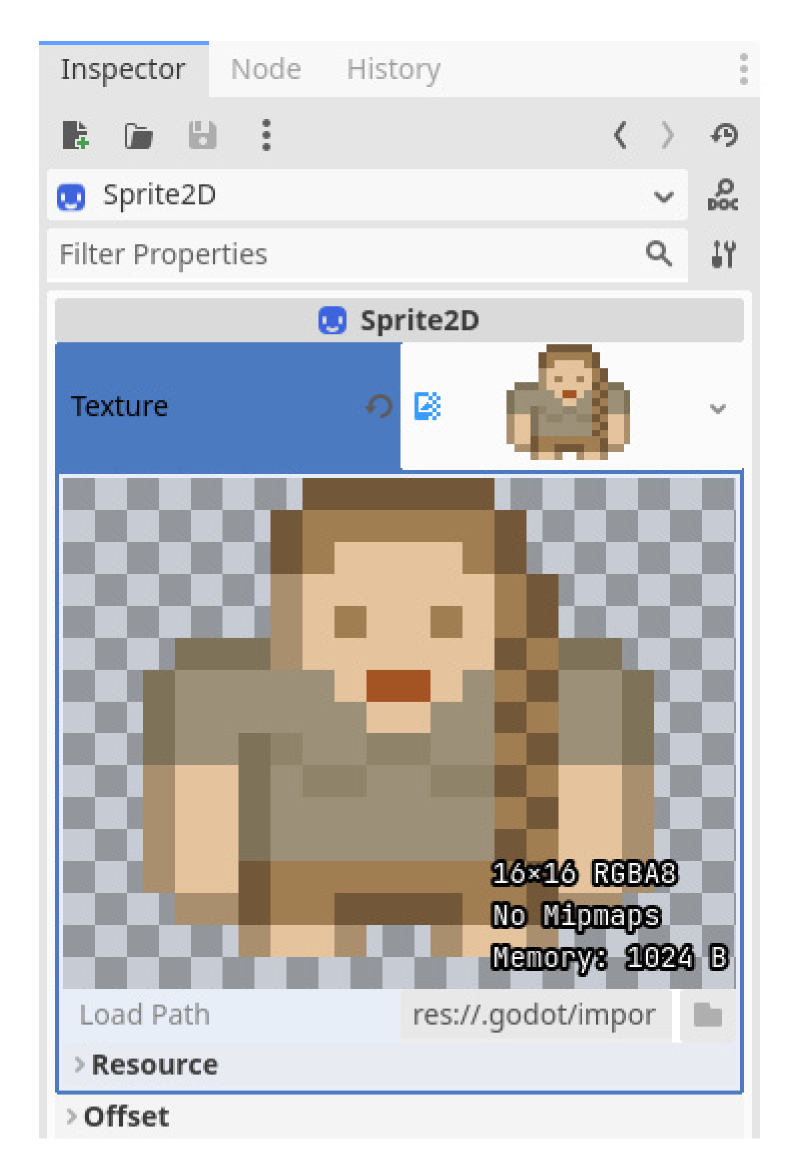


Рисунок 6.5 – Добавление текстуры к узлу спрайта

Спрайт также должен отобразиться на 2D-панели редактора. Однако он кажется очень маленьким. Это потому, что изображение имеет размер всего **16** × **16** пикселей. Давайте немного его увеличим. На вкладке **Transform** панели **Inspector** для спрайта установите **Масштаб (Scale)** на **3**. Вы можете задать масштаб для осей X и Y отдельно, но мы хотим, чтобы

они обе были равны, чтобы спрайт масштабировался без растяжения:

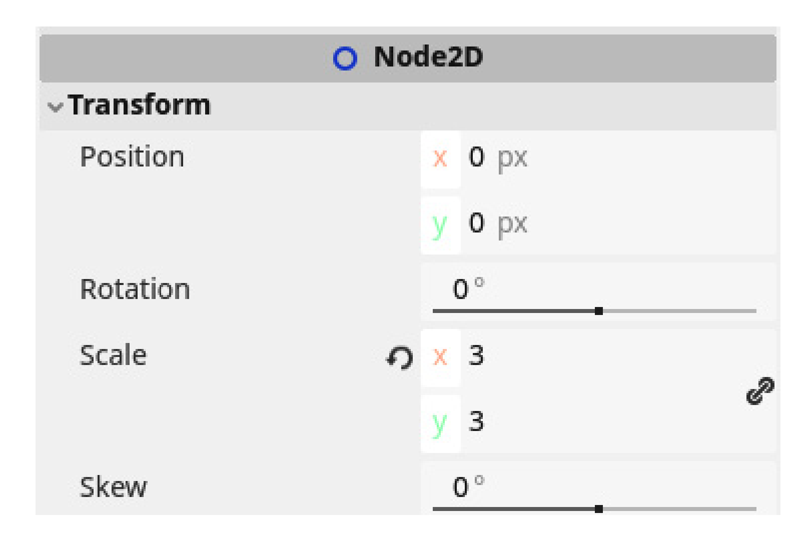


Рисунок 6.6 – Свойства преобразования узла спрайта

О нет – что это?

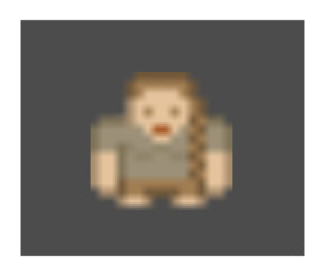


Рисунок 6.7 – Размытый пиксельный спрайт

Спрайт выглядит размытым! Это произошло из-за того, что мы используем ресурсы**пиксель-арт**, стиль, известный своими блочными пикселями. При масштабировании Godot Engine использует алгоритм, который размывает эти пиксели. Это отлично подходит для других стилей искусства, таких как рисованное или векторное искусство, но не для пиксельного искусства. К счастью, есть решение. Выполните следующие действия:

1. Перейдите в **Проект (Project)** | **Настройки проекта**

(Project Settings):

Scene Project Debug Editor Help

Scene

Project Settings...

Version Control

Export...

Install Android Build Template...

Open User Data Folder

Customize Engine Build Configuration...

Tools

Reload Current Project

Quit to Project List Ctrl+Shift+Q

Рисунок 6.8 – Переход к настройкам проекта...

1. В разделе **Rendering | Textures** установите **Фильтр текстур по умолчанию (Default Texture Filter)** на **Ближайший (Nearest):**

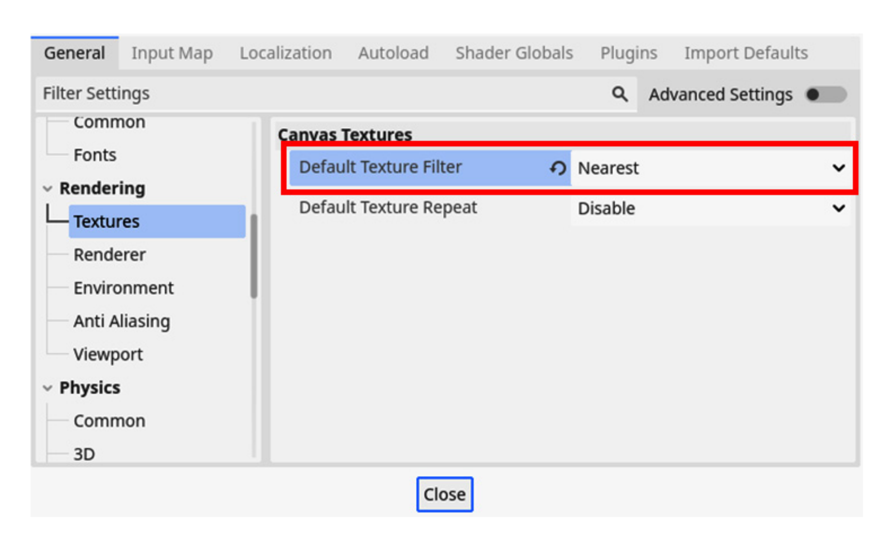


Рисунок 6.9 – Установка фильтра текстуры по умолчанию на ближайший

Эти настройки масштабируют изображение таким образом, чтобы оно лучше подходило для пиксельной графики. Теперь наш спрайт выглядит намного лучше!

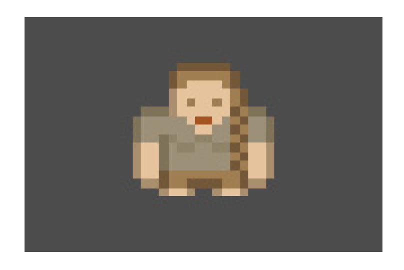


Рисунок 6.10 – Чёткий пиксельный спрайт

Теперь, когда мы видим нашего персонажа игрока, давайте рассмотрим отображениепользовательского интерфейса (UI) здоровья.

Отображение здоровья

Давайте добавим что-нибудь для отображения здоровья игрока над персонажем. Конечно, мы пока не создали скрипт для игрока, который отслеживает здоровье, но мы можем разместить визуальные эффекты на этом месте. Мы будем использовать узел **Label**, который может отображать текст в игре:

1. Найдите и добавьте узел **Label** к узлу **Player**.
2. Назовите узел **HealthLabel**:

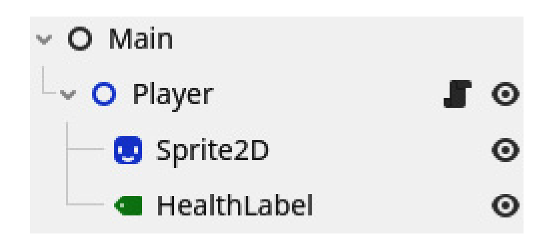


Рисунок 6.11 — Дерево сцены с добавленным узлом HealthLabel

1. При выборе узла **Label**, панель **Инспектор** будет содержать свойство **Text**. Введите в него **10/10**, как будто у игрока 10 из 10 жизней:

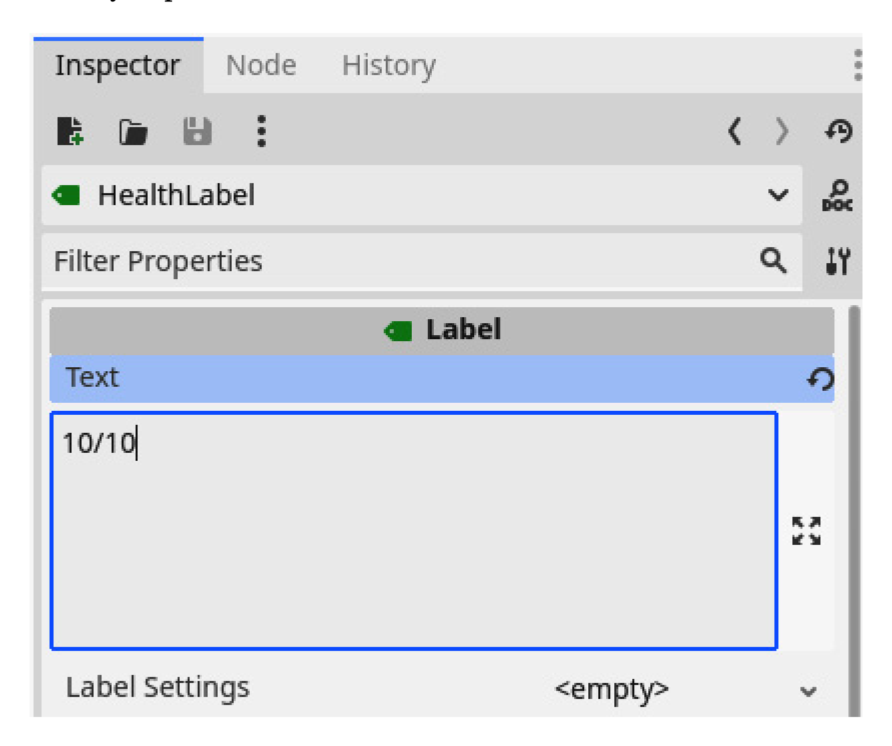


Рисунок 6.12 – Вид инспектора узла Label с текстом «10/10»

1. Затем расположите узел Label чуть выше спрайта персонажа, так, чтобы надпись не перекрывала спрайт:

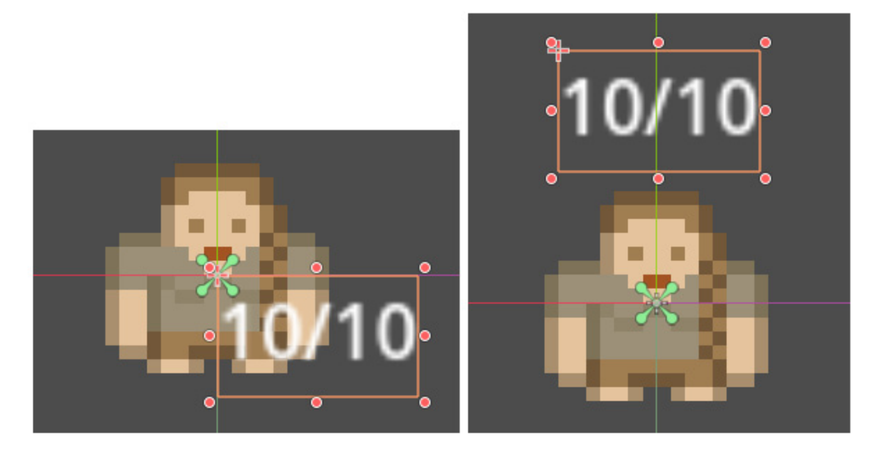


Рисунок 6.13 – Изменение расположения узла HealthLabel над персонажем игрока

Отлично! Теперь узел **HealthLabel** на своём месте. Мы сможем обновить его содержание позже и используем для этого скрипт (см. раздел *Создание скрипта игрока*). Этого нам пока достаточно для настройки узлов в дереве сцены.

Теперь давайте посмотрим, как мы можем манипулировать добавленными нами узлами.

Манипулирование узлами в редакторе

Теперь, когда у нас есть небольшое дерево сцены, давайте посмотрим, какие инструменты у нас есть для манипулирования узлами. Если вы посмотрите на правый верхний угол 2D-редактора, вы увидите некоторые из этих инструментов:

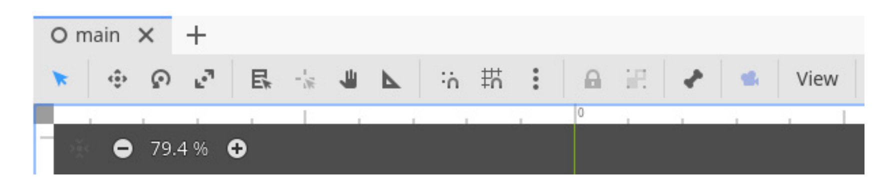


Рисунок 6.14 – Панель инструментов для режима 2D в редакторе

На этой панели есть много интересных инструментов, но наиболее важны на данный момент первые четыре:

- **Режим выделения (Select mode)**: Это режим по умолчанию, и он является многофункциональным. Вы можете выбирать узлы в сцене и перетаскивать их.
- **Режим перемещения (Move mode)**: в этом режиме вы можете перемещать выбранный узел.
- **Режим вращения (Rotate mode)**: в этом режиме вы можете вращать выбранный узел.
- **Режим масштабирования (Scale mode)**: в этом режиме вы можете масштабировать выбранный узел.

Попробуйте эти режимы, выбрав узел **Player** и немного повозившись. Это может привести вас к такому результату:

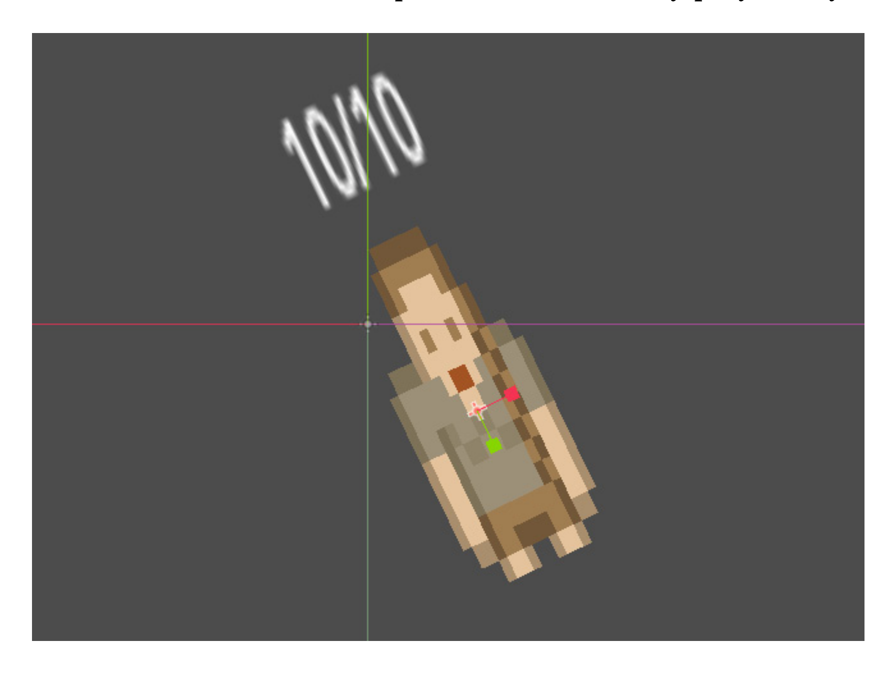


Рисунок 6.15 – Узел Player после множественных манипуляций и преобразований

Вы должны заметить, что когда вы перемещаете, вращаете или масштабируете узел, его узлы-потомки будут изменяться таким же образом. Это наследование трансформации является силой иерархической системы узлов.

Если вы посмотрите на вкладку **Transform** в доке **Инспектор (Inspector)** узла **Player**, вы увидите точные изменения, которые вы внесли в него. Если вы измените любое из этих значений, вы увидите что это изменение немедленно отобразится в 2D-редакторе:

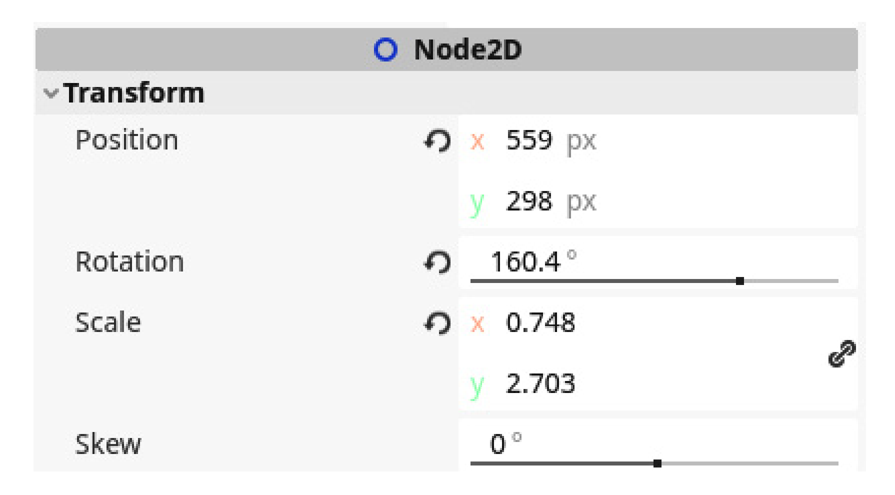


Рисунок 6.16 – Параметры Transform после множественных манипуляций преобразования

В качестве эксперимента попробуйте изменить значение параметра перекоса **Skew** В доке **Инспектор**.

Прежде чем продолжить работу со следующими разделами, не забудьте сбросить все эти манипуляции в доке **Transform** узла **Player**. Вы можете сделать это, просто нажав на символ ↺ рядом с каждым свойством. Эта кнопка вернет свойство к значению по умолчанию. Давайте также установим положение узла **Player** так, чтобы персонаж игрока был примерно по центру экрана:

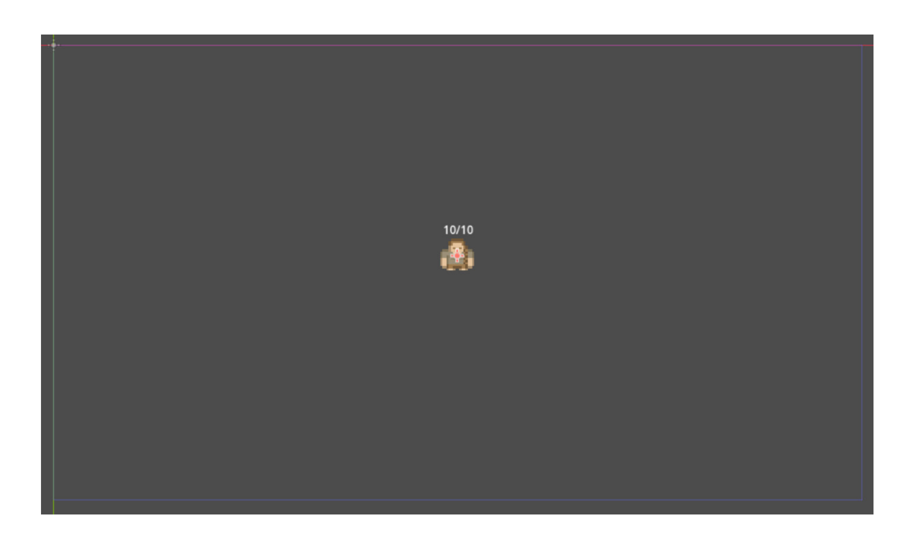


Рисунок 6.17 – Расположение персонажа игрока в центре экрана

На этом мы закончили создание базы для нашего игрового персонажа и узнали, как манипулировать узлами в редакторе. Далее мы сосредоточимся на сценарии игрового персонажа и узнаем, как можно манипулировать узлами через код.

Создание скрипта игрока

Настал момент, к которому мы готовились. Мы уже знаем, как это сделать! Итак, начнём с создания нового скрипта, который будет прикреплен к узлу **Player**:

1. Щелкните правой кнопкой мыши по узлу **Player** и выберите **Впикрепить скрипт... (Attach Script...)**:

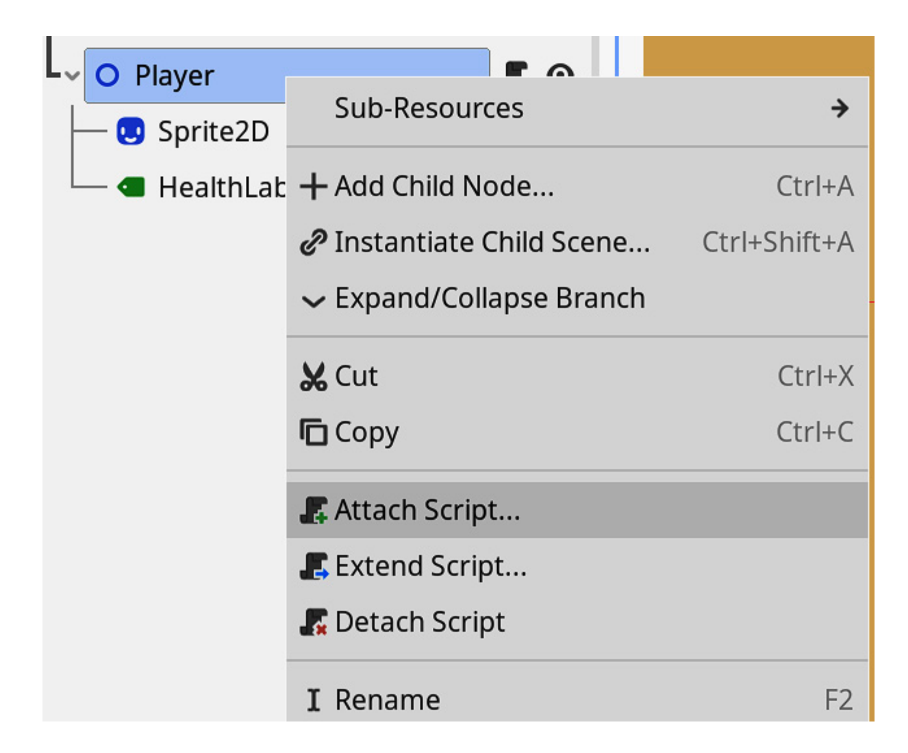


Рисунок 6.18 – Прикрепление скрипта к узлу Player

1. В появившемся диалоговом окне выберите скрипт **player.gd**:

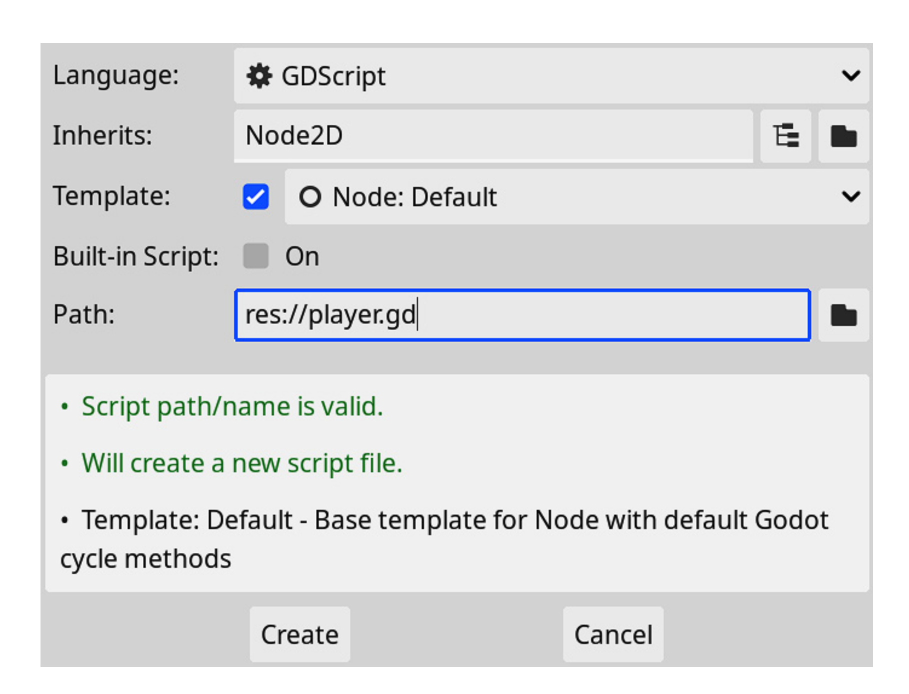


Рисунок 6.19 – Создание скрипта player.gd

1. Пока что мы не будем слишком усложнять его и и просто добавим немного кода для управления здоровьем игрока:

   ```
   extends Node2D
   const MAX_HEALTH: int = 10
   var health: int = 10
   func add__health_points(difference: int):
      health += difference
      health = clamp(health, 0, MAX_HEALTH)
   ```

   Функция **clamp()**, которую мы используем в функции **add_health_points()**, принимает числовое значение в качестве первого аргумента и сохраняет его между двумя вторыми числовыми параметрами.

Таким образом, здоровье всегда находится в диапазоне от **0** до **MAX_HEALTH**, что обеспечивает максимальное значение **10**.

Важное примечание

Помните, что вы можете нажать *Ctrl* и щёлкнуть в Windows и Linux или нажать *Option* и щёлкнуть в Mac по любой функции, чтобы перейти к документации и посмотреть, что она делает.

Теперь мы можем изменить здоровье игрока. давайте посмотрим, как мы можем обновить узел **HealthLabel**, который мы создали ранее, чтобы отразить это значение.

Ссылки на узлы в скрипте

Мы хотим обновить содержимое узла **HealthLabel** нашего игрового персонажа в соответствии с количеством здоровья, которое у игрока осталось. Чтобы изменить узлы в нашей сцене из скрипта, нам нужно иметь возможность ссылаться на них. К счастью, в Godot 4 это довольно просто.

Есть несколько способов получить ссылку на узел, но самый простой — это долларовая нотация. Эта нотация выглядит так:

```
$HealthLabel
```

Нотация начинается со знака доллара (**$**), за которым следует путь по дереву сцены к нужному нам узлу. Здесь мы ссылаемся на узел HealthLabel - метку здоровья, которую мы создали ранее.

Обратите внимание, что это относительно узла со скриптом, где этот путь упоминается. Так что если скрипт прикреплён к основному узлу и мы хотим сослаться на метку здоровья игрока, нотация будет выглядеть так:

```
$Player/HealthLabel
```

Итак, теперь, когда мы знаем, как получить ссылку на узел, давайте создадим небольшую функцию, которая обновляет метку здоровья игрока, и вызовем её в функции **add_health_points()** function:

```
func update_health_label():
   $HealthLabel.text = str(health) + "/" + str(MAX_HEALT
```

В функции **update_health_label()** мы берём узел **HealthLabel** и напрямую изменяем его переменную **text**. Это изменит текст, отображаемый меткой на экране.

Здесь мы используем новую функцию **str()** в **update_health_label()**. Эта функция принимает любой параметр и преобразует его в строку. Нам нужно это сделать, потому что знак + может объединять только строки, поэтому для объединения значений **health** и **MAX_HEALTH**, которые являются целыми числами, нам придется преобразовать их в строку.

Теперь мы можем использовать эту функцию **update_health_label()** всякий раз, когда изменяем значение здоровья — **health**:

```
func add_health_points(difference: int):
   health += difference
   health = clamp(health, 0, MAX_HEALTH)
   update_health_label()
```

Точно так же мы можем напрямую изменить то, что отображает узел **HealthLabel**. Но есть лучший способ доступа или ссылки на узлы в дереве сцены: путем их кэширования. Мы рассмотрим это далее.

Кэширование ссылок на узлы

Хотя долларовая нотация очень удобна, иногда вам нужно будет часто обращаться к определённому узлу. В таких случаях использование долларовой нотации будет медленным, поскольку движку придётся продолжать искать узел в дереве и обращаться к нему каждый раз.

Кэширование

На компьютерном языке кэширование означает сохранение

определённого фрагмента данных для последующего использования, чтобы вам не приходилось загружать его каждый раз, когда он понадобится.

Чтобы не искать узел каждый раз, мы можем сохранить ссылку на узел в переменной. Например, мы можем изменить скрипт игрока следующим образом:

```
extends Node2D
@onready var _health_label: Label = $HealthLabel
func update_health_label():
   _health_label.text = str(health) + "/" + str(MAX_HEAL
```

Здесь вы видите, что мы сохраняем ссылку на узел **HealthLabel** в переменной с именем **_health_label**. Позже мы можем использовать эту ссылку.

Плюс в том, что нам нужно изменить путь к узлу только в одной точке: в строке, где ссылка сохраняется в переменной. Другой плюс в том, что мы можем указать тип переменной с типом узла. Таким образом, мы делаем его ещё безопаснее, чем предыдущий способ ссылки на узел.

Вы также заметите, что я использую аннотацию **@onready**. Мы вызываем команды, которые начинаются с аннотации **@**, как показано ранее. Эта аннотация выполняет эту строку кода, когда узел готов и вошел в дерево сцены. Это происходит прямо перед вызовом функции **_ready()** этого узла. В Godot функция **_ready()** каждого узла вызывается после того, как каждый из его дочерних узлов готов, что означает, что их функции **_ready()** вызываются до функции **_ready()** родительского узла. Нам нужно дождаться этого момента, чтобы получить какие-либо узлы в дереве, потому что в противном случае есть вероятность, что они еще не существуют!

Аннотации

Есть ещё аннотации. Мы вернемся к ним, когда будем их использовать. Но уже сейчас надо помнить, что аннотации

влияют на то, как внешние инструменты будут обрабатывать скрипт, и не влияют на логику внутри самого скрипта.

Я советую вам всегда кэшировать переменные, как описано здесь, поскольку это сохранит ваш код чистым и быстрым.

Тестируем скрипт игрока

Чтобы опробовать то, что мы создали на данный момент, мы можем запустить быстрый тест, добавив функцию **_ready()** в скрипт plигрока:

```
func _ready():
   add_health_points(-2)
```

Теперь, когда вы запустите сцену, вы должны увидеть, что метка здоровья, HealthLabel, показывает **8/10**, вот так:



Рисунок 6.20 – Метка здоровья игрока теперь показывает 8/10

После тестирования скрипта снова удалите предыдущие строки, чтобы они не мешали остальной части нашего кода.

В этом разделе мы узнали, как ссылаться на узлы из дерева сцены в нашем коде и как обновлять значения этих узлов. Мы также настроили базовый скрипт для отслеживания здоровья нашего игрока. В следующем разделе мы узнаем об экспорте переменных.

Экспорт переменных в редактор

Мы всегда определяли переменные в коде, и каждый раз, когда мы хотели их изменить, нам приходилось менять и код. Но в Godot легко выставить переменные в редакторе, чтобы мы могли изменять их, даже не открывая редактор кода. Это чрезвычайно полезно, когда вы хотите что-то протестировать и настраивать переменные на лету. Экспортированная переменная появляется в доке **Инспектор** этого узла, как и свойства transformation и text которые мы видели в разделе *Манипулирование узлами в редакторе*.

Важное примечание

Экспортированная переменная также полезна для людей, которые не умеют писать код, например, дизайнеров уровней, но всё же хотят изменить поведение определённых узлов.

Чтобы экспортировать переменную в редактор, мы можем использовать аннотацию **@export**. Давайте изменим строку, где мы определяем переменную **health**, вот так:

```
@export var health: int = 10
```

Обязательно сохраните скрипт. Перейдите в режим 2D с помощью кнопки в верхней части редактора.

2D 3D Script AssetLib

Рисунок 6.21 – Нажмите 2D, чтобы перейти в 2D-редактор

Щёлкните по нашему узлу **Player**, и вы увидите переменную **health** в доке **Inspector**. Это наша экспортированная переменная. Её изменение изменит значение переменной в начале игры, без редактирования скрипта напрямую:

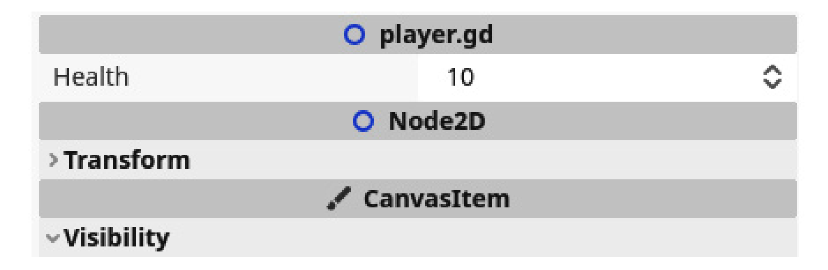


Рисунок 6.22 – Переменная health как экспортированная переменная в доке Инспектор узла Player

Теперь, когда вы измените значение здоровья игрока через док **Inspector** и запустите игру, вы увидите, что она не отображает правильное значение. Всё, что нам нужно сделать, чтобы решить эту проблему, это обновить метку **здоровья** в функции **_ready()**, вот так:

```
func _ready():
   update_health_label()
```

Это обеспечит обновление метки здоровья с момента попадания узла **Player** в дерево сцены.

Дополнительная информация

Если вы хотите узнать больше об экспортируемых переменных, вы можете ознакомиться с официальной документацией: https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_exports.html.

Теперь мы начинаем игру с правильным количеством здоровья, отображаемым на метке здоровья. Но есть лучший способ обновления этой метки здоровья: использование сеттеров и геттеров.

Сеттеры и геттеры

Когда вы измените значение здоровья игрока через док **Инспектор** и запустите игру, вы увидите, что значение не обновляется. Но правильное значение будет выведено, если вы

запустите его с функцией **_ready()**, например так:

```
func _ready():
   print(health)
```

Это происходит потому, что функция **update_health_label()** не вызывается, когда мы меняем значение!

К счастью, мы можем это исправить. В программировании существуют функции **getter** и **setter**. Эти функции вызываются, когда вы получаете или устанавливаете значение переменной. С помощью этих функций получения или установки мы можем выполнить всю логику, необходимую для обработки нового значения. Мы можем определить функцию получения и установки для нашей переменной здоровья, health, следующим образом:

```
@export var health: int = 10:
   get:
      return health
   set(new_value):
      health = clamp(new_value, 0, MAX_HEALTH)
      update_health_label()
```

Итак, геттер определяется ключевым словом **get:**, за которым следует блок кода, определяющий логику геттера, а сеттер определён строкой **set(new_value):**, за которым следует его блок кода. **new_value** — это новое значение, которое присваивается переменной health. В сеттере мы получаем возможность обработать это значение при необходимости или запустить другие процессы. В нашем случае мы не хотим обрабатывать новое значение, но хотим обновить метку здоровья.

Геттер не делает ничего особенного — он просто возвращает значение здоровья. С другой стороны, сеттер фиксирует (clamp) новое значение так, чтобы оно находилось в диапазоне допустимых значений, а затем обновляет метку здоровья.

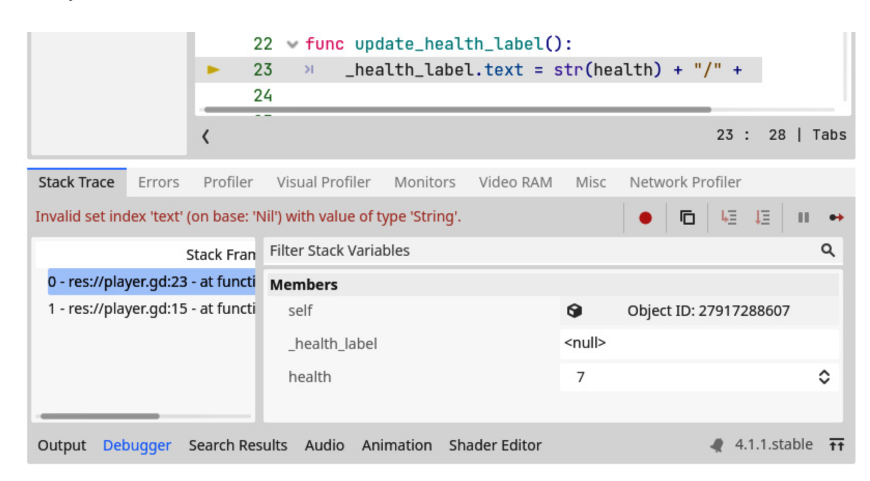
Когда мы получаем или устанавливаем значение здоровья — **health**, интерпретатор сначала выполнит эти функции. Вот пример:

```
print(health) # Выполнить геттер
health = 100 # Выполнить сеттер
```

Это также упрощает функцию **add_health_points()**, ведь нам больше не нужно фиксировать новое значение здоровья в заданном диапазоне, поскольку это уже сделано в сеттере. Итак, давайте обновим функцию **add_health_points()** следующим образом:

```
func add_health_points(difference: int):
   health += difference
```

Но что это? Теперь проект выдаёт ошибки, когда мы его запускаем!



="Рисунок 6.23 – Ошибка, показывающая, что узел метки здоровья не существует

Функция setter выполняется до создания ссылки **_health_label**. Поэтому мы должны убедиться, что у ссылки **_health_label** есть значение, прежде чем обновлять её текст. Если это не так, мы можем просто вернуться из функции:

```
func update_health_label():
    if not is_instance_valid(_health_label):
        return
    _health_label.text = str(health) + "/" + str(MAX_HEAL
```

Функция **is_instance_valid()** проверяет, является ли ссылка на узел действительной. Она возвращает **true**, если это так, и **false** в противном случае.

Проверка существования ссылки на узел

Первым побуждением может быть проверка того, не является ли ссылка на узел **нулевой (null)** выполнив **_health_label !=** **null**. Однако это не гарантирует, что узел доступен. Например, когда узел удаляется, эта проверка на **null** всё равно вернёт **true**, поскольку ссылка всё ещё существует в переменной. **is_instance_valid(_health_label)** проверит не только то, является ли переменная **нулевой**, но и то, что узел всё ещё существует и используется в дереве сцены.

На этом этапе код для плеера должен выглядеть так:

```
extends Node2D
const MAX_HEALTH: int = 10
@onready var _health_label: Label = $HealthLabel
@export var health: int = 10:
    get:
        return health
    set(new_value):
        health = clamp(new_value, 0, MAX_HEALTH)
        update_health_label()
func _ready():
    update_health_label()
func update_health_label():
    if not is_instance_valid(_health_label):
        return
    _health_label.text = str(health) + "/" + str(MAX_HEAL
func add_health_points(difference: int):
    health += difference
```

Сеттеры и геттеры помогают нам инкапсулировать поведение, связанное с обновлением переменных, как было показано в *Главе 5*. Это абстрагирует логику того, что должно произойти при обновлении этой переменной, и пользователю класса не нужно больше беспокоиться об этом.

При такой настройке кода здоровье (health) нашего игрока можно легко обновить с помощью обычных или специальных операторов назначения, и метка здоровья (HealthLabel) обновится соответствующим образом.

Изменение значений во время игры

Ещё одна крутая вещь об этих экспортируемых переменных — теперь, когда у нас есть сеттер и геттер, определённые для них, мы можем изменять их во время работы игры! Таким образом, если вы запустите игру и измените параметр **health** в доке **Инспектор** во время её работы, вы увидите, что это изменение мгновенно отразится на метке здоровья.

Это, как правило, работает со всеми встроенными параметрами! Если вы держите игру открытой и меняете, например, параметры **Transformation** у игрока, , вы увидите, как они меняются в реальном времени.

Это пригодится нам в дальнейшем, чтобы не перезапускать игру во время работы над ней.

Различные типы экспортируемых переменных

При экспорте переменной, с указанием типа, Godot выберет правильный тип поля ввода для этого типа. Он установит числовое поле ввода со стрелками вверх и вниз для целых чисел, а для строк будет использовать обычный текстовый ввод, например:

```
@export var health: int = 10
```

```
@export var damage: float = 0.0
@export var player_name: String = "Эрика"
```

Эти три строки экспортируют переменные в редактор, но каждая из них со своим типом данных: целое число, число с плавающей точкой и строка соответственно. В результате мы получаем разные типы полей ввода для каждой из переменных:

Рисунок 6.24 – Различные типы экспортируемых переменных

Необходимо упомянуть, что есть и другие аннотации экспорта. Одна из них — аннотация **@export_range**, которая указывает диапазон чисел, в котором должно находиться значение, например:

```
@export_range(0, 10) health: int = 10
```

В этом фрагменте кода мы экспортируем переменную **health** и указываем, что она должна быть в числом в диапазоне от 0 до 10, включая граничные значения — 0 и 10. Когда вы опробуете этот диапазонный экспорт, вы увидите, что нельзя вводить значения, которые выходят за пределы этого диапазона.

Чтобы сделать его более динамичным, мы можем использовать переменную **MAX_HEALTH** которую мы определили ранее в скрипте player.gd:

```
@export_range(0, MAX_HEALTH) health: int = 10
```

Экспорт переменных — очень важный метод, который нужно сохранить в нашем наборе инструментов для настройки переменных и значений при тестировании игры. Теперь давайте

обратим внимание на арену и мир, по которым будет ходить игрок.

Создание маленького мира

Теперь, когда у нас есть маленький игровой персонаж, давайте создадим мир, в котором он будет жить! В этом разделе мы подробно рассмотрим арену, на которой игроку предстоит сражаться со сложными противниками.

Изменение цвета фона

Давайте начнём с простого, изменив цвет фона для нашей арены. Мы можем легко сделать это из настроек проекта:

1. Перейдите в раздел **Рендеринг (Rendering)** | **Окружение (Environment)** в настройках проекта:

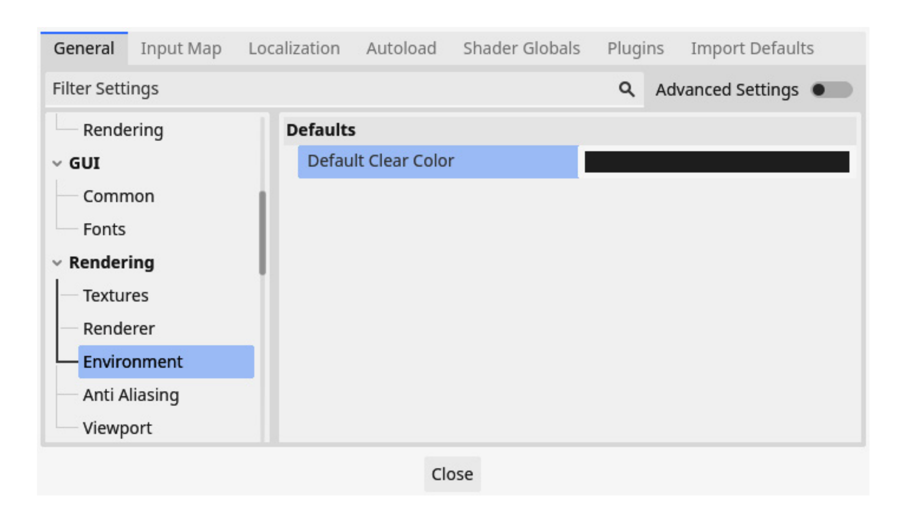


Рисунок 6.25 – Поиск параметра «Цвет фона по умолчанию» в разделе «Рендеринг» | «Окружение» в настройках проекта

1. Установите в параметре **Цвет фона по умолчанию (Default Clear Color)** подходящий цвет. Я выбрал **#e0bf7b**, потому что он похож на песок или засохшую грязь:

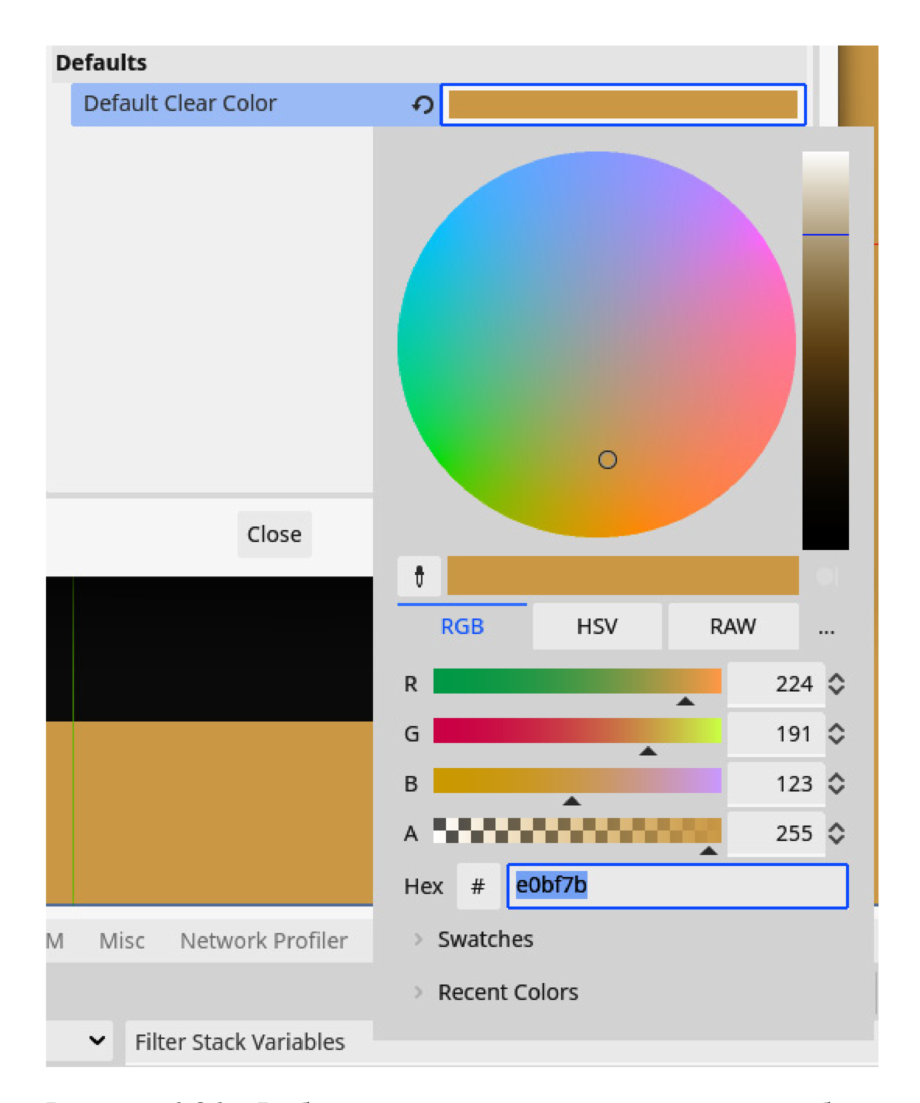


Рисунок 6.26 – Выбор цвета с помощью инструмента выбора цвета

Теперь, когда этот красивый цвет готов , давайте добавим некоторые детали, такие как валуны и стены, на нашу арену.

Добавление валунов с помощью узла Polygon2D

Теперь, когда у нас есть земля, на которой может стоять персонаж игрока, давайте добавим несколько валунов, которые будут служить препятствиями на арене. Для этого мы будем использовать узел **Polygon2D**. Этот узел может нарисовать на экране любую многоугольную форму любого цвета:

1. Добавьте узел **Node2D** с именем **Arena** к корневому узлу нашей сцены **Main**.
2. Теперь перетащите узел **Arena**, который мы только что создали, выше узла **Player** в дереве сцены. Это гарантирует, что всё внутри узла **Arena** будет отрисовано позади узла **Player**. Подробнее об этом читайте в разделе *Порядок отрисовки узлов*.
3. Мы поместим все элементы арены, такие как валуны и стены, в этот узел. Таким образом, мы сохраним структуру дерева красивой и аккуратной.
4. Теперь добавьте узел **Polygon2D** под узлом **Arena** и назовите его **Boulder**:

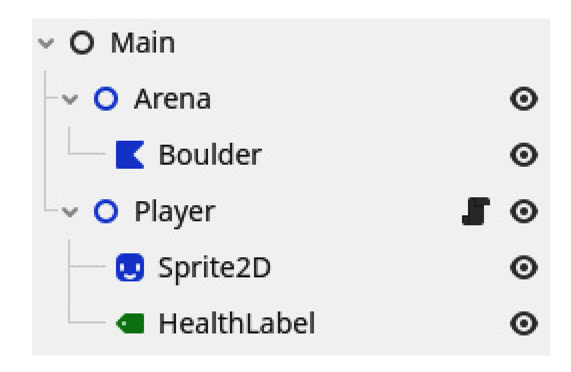


Рисунок 6.27 – Дерево сцены с узлами «Arena» и узлом «Boulder»

1. Вы можете добавлять точки в многоугольник, щёлкнув левой кнопкой мыши в любом месте экрана, когда выбран узел **Boulder**. Щелчок правой кнопкой мыши удалит точку. Вы также можете перетаскивать ранее размещенные точки. Разместите несколько точек и замкните форму, щёлкнув по первой размещенной точке:

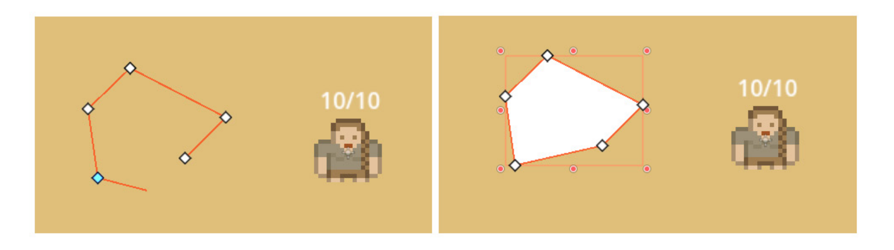


Рисунок 6.28 – Рисование валуна с использованием узла Polygon2D

1. Задайте в свойстве цвета узла **Boulder** что-то похожее на камень. Я выбрал **#504f51**.

Эти валуны могут показаться простыми, но они послужат нашей цели.

Порядок отрисовки узлов

Итак, почему мы перетащили узел **Arena** выше узла **Player** в дереве сцены? По умолчанию узлы рисуются в том порядке, в котором они находятся в дереве. Узлы, ближайшие к своим родителям, рисуются первыми, а те, которые дальше от родительского узла в структуре дерева, рисуются поверх тех, что ниже.

Есть способы обойти это, но это выходит за рамки этой книги. Поэтому сейчас мы должны правильно структурировать наше дерево узлов:

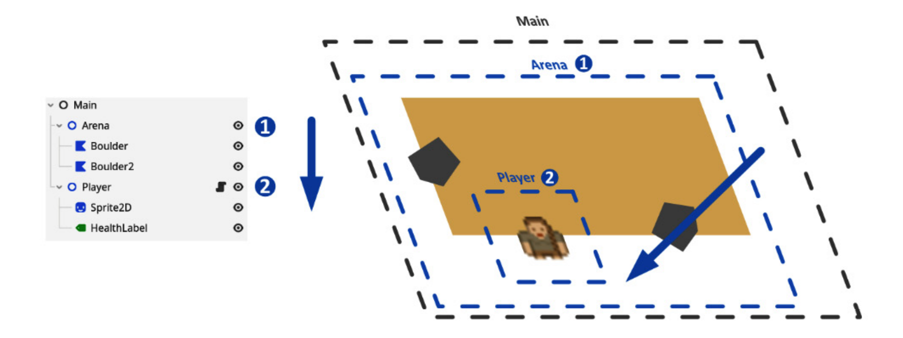


Рисунок 6.29 – Узлы рисуются в том порядке, в котором они

находятся в дереве сцены.

Хорошо структурированное дерево будет отображать все узлы именно в том порядке, в котором мы хотим.

Создание внешней стены

Для внешней стены арены мы снова воспользуемся узлом **Polygon2D**, но на этот раз другим способом:

1. Добавьте узел **Polygon2D** под узлом **Arena** и назовите его **OuterWall**.
2. Нарисуйте грубый прямоугольник, который будет внутренней частью арены. Ничего страшного, если этот прямоугольник не идеален. Это придаст арене еще более средневековый вид:

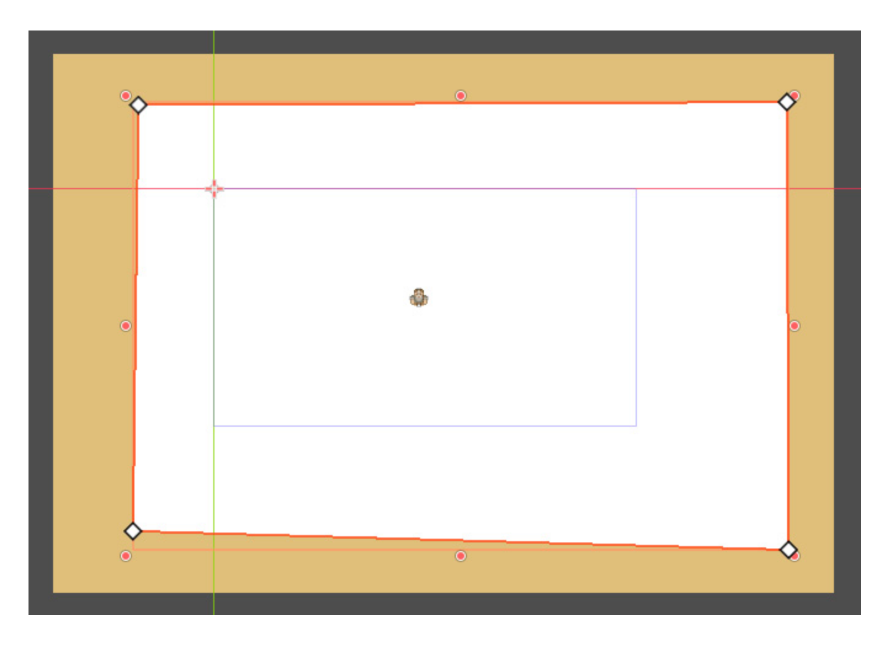


Рисунок 6.30 – Рисование внешней стены арены с использованием узла Polygon2D

1. Теперь, выбрав **OuterWall**, найдите и включите параметр **Invert** в доке **Inspector**. Эта опция инвертирует форму и делает её похожей на внешние стены арены:

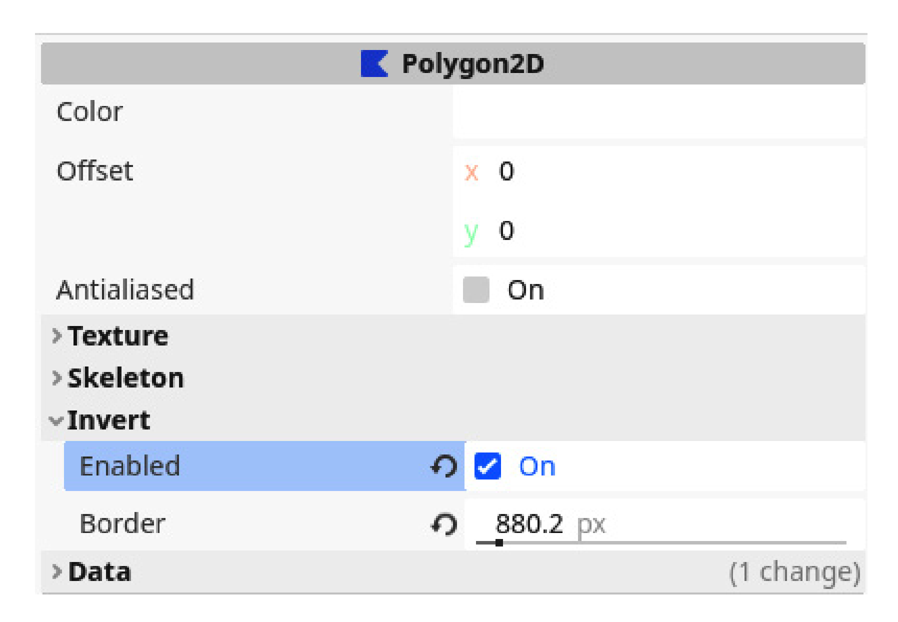


Рисунок 6.31 – Инвертирование формы узла Polygon2D

1. Задайте для свойства **Border** в разделе **Invert** большое значение, например **1000px**, чтобы стены расширялись очень далеко.
2. Задайте стене подходящий цвет. Я выбрал **#2d2c2e**, который немного темнее валунов, чтобы игрок видел разницу:

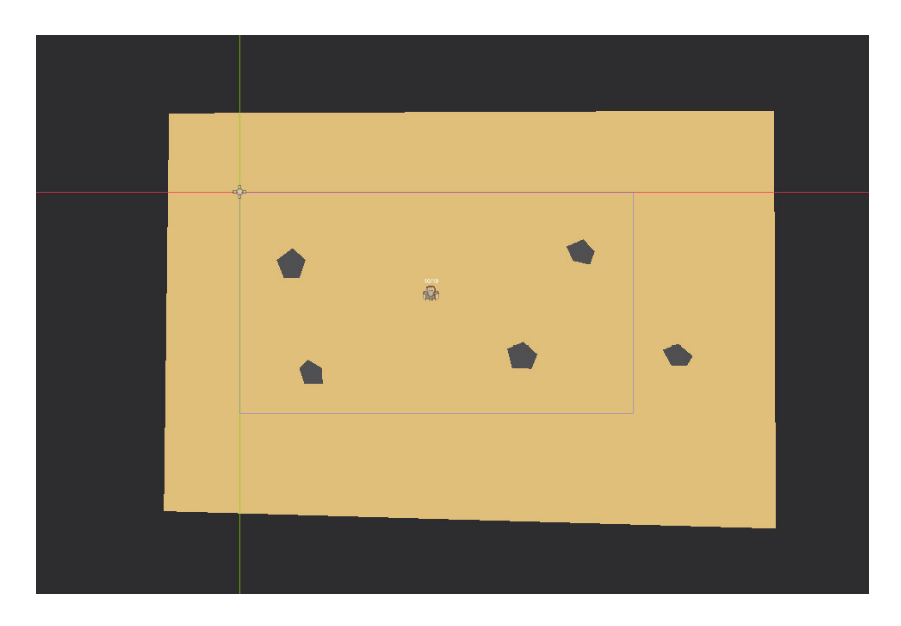


Рисунок 6.32 – Получившаяся арена

Инвертирование полигона позволяет очень легко создать внутреннее пространство арены или комнаты. Естественным следующим шагом будет дать волю воображению и создать визуально красивую арену.

Используем творческий подход

С помощью этих простых инструментов проявите выдумку и создайте интересную местность, которая может служить ареной.

Например, вы можете добавить ещё несколько валунов на свою арену. Вы можете сделать это, создав совершенно новый узел **Polygon2D** или скопировав свой предыдущий валун и немного изменив его, перетаскивая точки и используя инструменты **Transform**, о которых мы узнали.

Вы также можете добавить больше стен и ещё немного изменить внешние границы арены.

Я придумал такую арену:

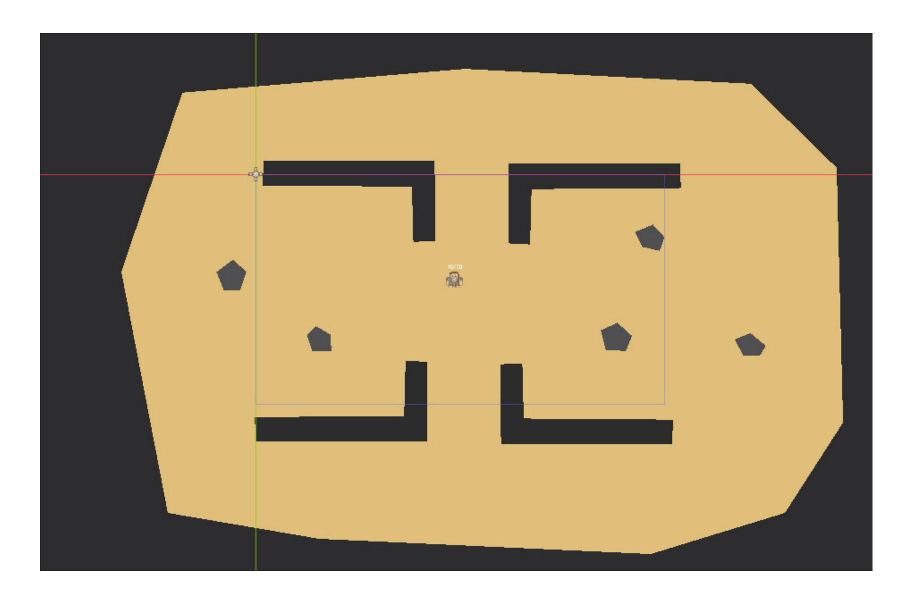


Рисунок 6.33 – Моя арена после того, как я потратил ещё некоторое время на её улучшение

Теперь, когда мы можем создавать наши маленькие миры и арены с помощью цветных прямоугольников и полигонов, у нас есть всё, чтобы создать базовую визуальную структуру нашей игры. Далее мы сделаем несколько дополнительных упражнений и подведем итог этой главы.

Дополнительные упражнения – Заточка топора

1. Начните с создания большего количества валунов и стен на основе того, что мы узнали в предыдущем разделе, чтобы вы могли заполнить свою арену.
2. Основываясь на том, что вы узнали о значении здоровья игрока — **health**, добавьте в скрипт игрока экспортируемую переменную, которая отслеживает количество монет у игрока, называемую **number_of_coins**.
3. Добавьте сеттер и геттер для переменной **number_of_coins**.
4. Наконец, создайте узел метки - Label, которая показывает

монеты над головой игрока. Убедитесь, что все обрабатывается и обновляется правильно, чтобы мы могли обновить переменную из редактора и кода, а метка всегда оставалась актуальной при запуске игры.

Итоги

В этой главе мы создали нашу первую настоящую сцену. Мы увидели свойства и возможности некоторых узлов и расширили **Node2D** скриптом, который будет управлять здоровьем игрока. Мы также создали область, в которой будут происходить все действия.

В следующей главе мы сделаем возможным перемещение игрока, а также обновим наши познания в векторной математике. Не волнуйтесь – это не будет больно. Мы используем лишь небольшое её количество.

Опрос

- Почему мы начали с создания дизайн-документа игры (GDD), а не сразу перешли к её созданию?
- Как ссылаться на узлы в скрипте?
- Какое ключевое слово можно использовать, чтобы сделать переменную, например, количество здоровья — health, доступной в доке **Инспектор**?
- Для чего используются функции сеттер и геттер (setter) и (getter)?

7

Заставим персонажа двигаться

Если я скажу **физический движок**, у вас перед глазами наверняка промелькнут образы ученых, которые моделируют всевозможные явления реального мира и пытаются предсказать, что произойдет в реальном мире, — возможно, для изучения автокатастроф, погоды или оптимизации ветровых электростанций.

Игры тоже используют физический движок. Однако в играх этот движок имеет другие приоритеты. В отличие от тех, которые используют ученые, физика в играх не должна быть точной на 100%. Она просто должна давать ощущение реализма или, наоборот, его отсутствия.

В этой главе мы узнаем, как использовать физический движок Godot, чтобы нам не приходилось иметь дело со сложными взаимодействиями, такими как перемещение персонажей или столкновения между несколькими физическими объектами в мире.

Сначала нам придется сделать крюк и освежить наши основы в двумерной векторной математике. Обещаю, немного математики не повредит!

В этой главе мы рассмотрим следующие основные темы:

- Векторная 2D-математика
- Физический движок
- Ввод с клавиатуры и контроллера
- Отладка живой игры

Технические требования

Как и для каждой главы, окончательный код можно найти в репозитории GitHub в подпапке для этой главы:

https://github.com/PacktPublishing/Learning-GDScript-by-Developing-a-Game-with-Godot-4/tree/main/chapter07.

Освежим знания по векторной математике

Я знаю, что математика — это не первое, о чем люди думают, когда говорят о разработке игр, но правда в том, что это важная часть при работе с позициями, движением и физикой. Это отличный инструмент, который будет хорошо служить нам на протяжении всего пути разработки игр.

В этом разделе мы освежим свои знания о векторах и о том, как ими манипулировать.

Двумерная система координат

Давайте сначала рассмотрим, как мы определяем позиции в двумерном пространстве. Двумерное пространство определяется двумя осями: горизонтальной осью, именуемой *x* и вертикальной осью, именуемой *y*. Каждая точка в пространстве может быть выражена как значения по этим осям:

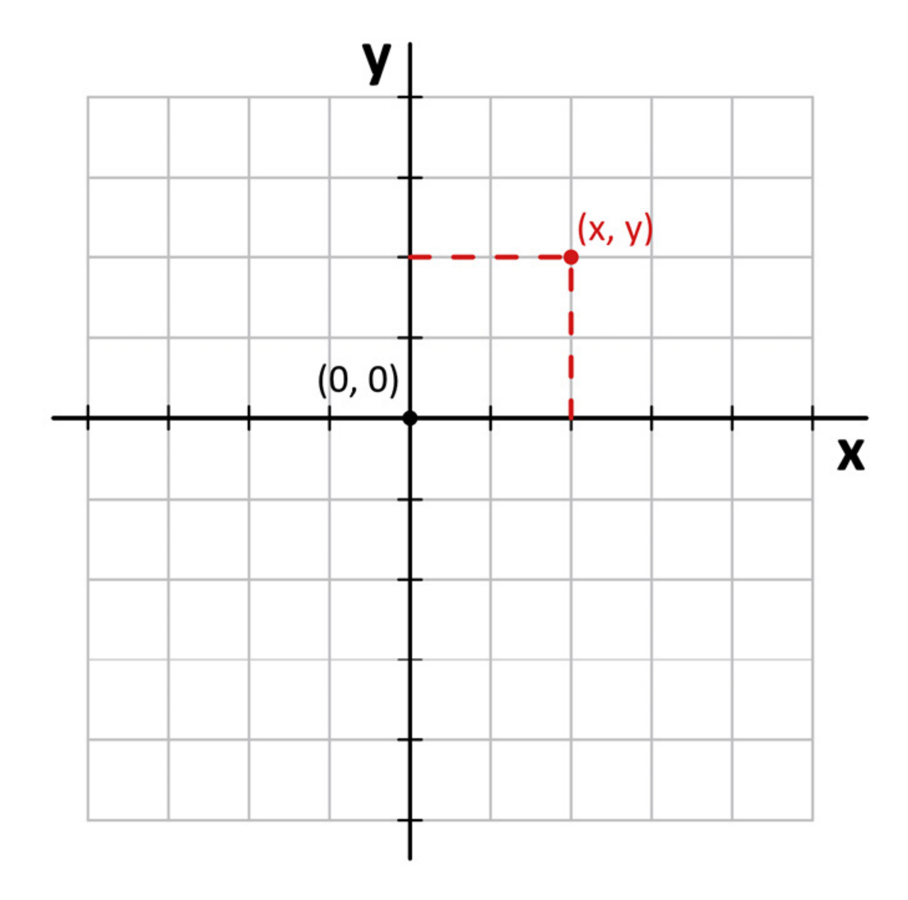


Рисунок 7.1 – Точка в двумерном пространстве определяется значениями x и y.

Математически и в этой книге мы записываем эти позиции как **(x, y)**. Первое значение в скобках — это значение *x*, а второе — значение *y*.

В школе вы, вероятно, узнали, что ось x положительна справа и отрицательна слева, в то время как ось y положительна при движении вверх и отрицательна вниз. Однако в компьютерной графике это не так! В Godot, как и почти в любом другом игровом движке и приложении компьютерной графики, ось y положительна при движении *вниз*:

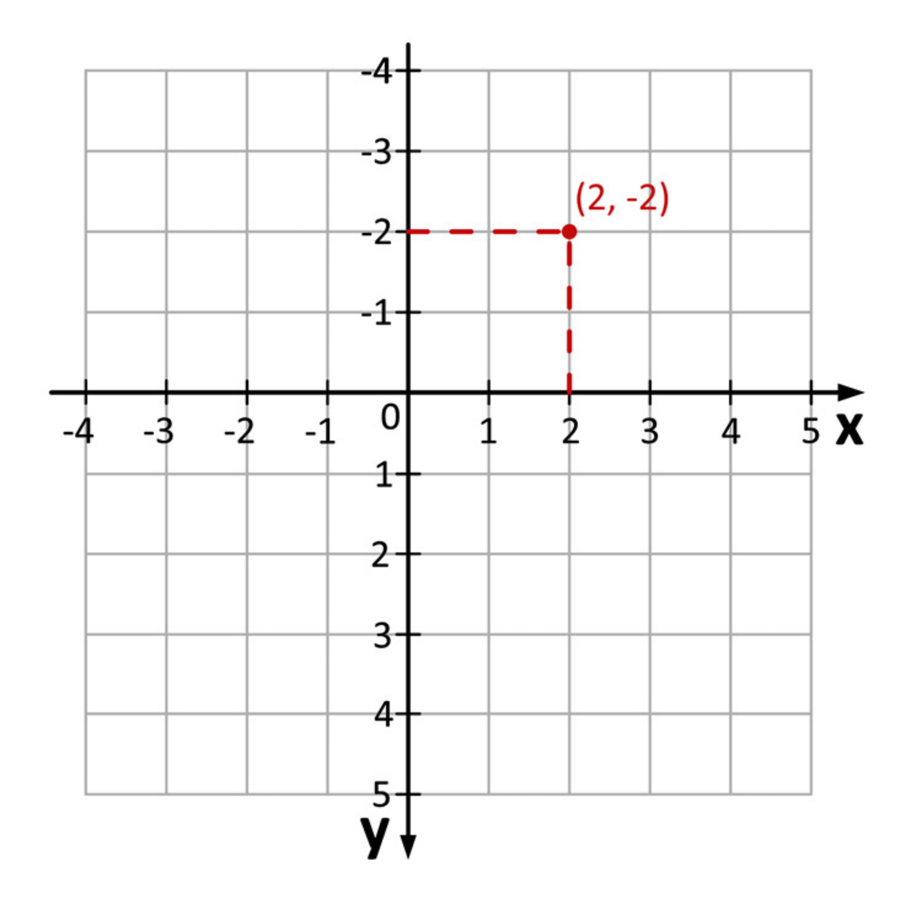


Рисунок 7.2 – Точка (2, -2) в движке Godot

Поначалу это может показаться нелогичным, но со временем вы это поймёте.

Важное примечание

Причина, по которой ось y направлена вниз, заключается в том, что компьютерная графика рассчитывалась таким образом, чтобы пиксель на экране, соответствующий $x = \mathbf{0}$ и $y = \mathbf{0}$, находился в верхнем левом углу экрана. Существует два распространённых объяснения:

Первые компьютеры часто отображали текст на основе латиницы, который шел слева направо и сверху вниз, поэтому символ (**0, 0**) располагался в левом верхнем углу.

Ранние мониторы представляли собой экраны на основе ЭЛТ, которые часто сканировали экран электронным лучом, начиная с верхнего левого угла и продвигаясь строка за строкой вниз.

Что такое вектор?

Что если мы представим позицию как смещение от точки (**0, 0**), т.е. начала координат? Ну, это то, что мы называем **вектором**! Вы можете представить их как стрелку, идущую от начала координат к положению, которое мы определили.

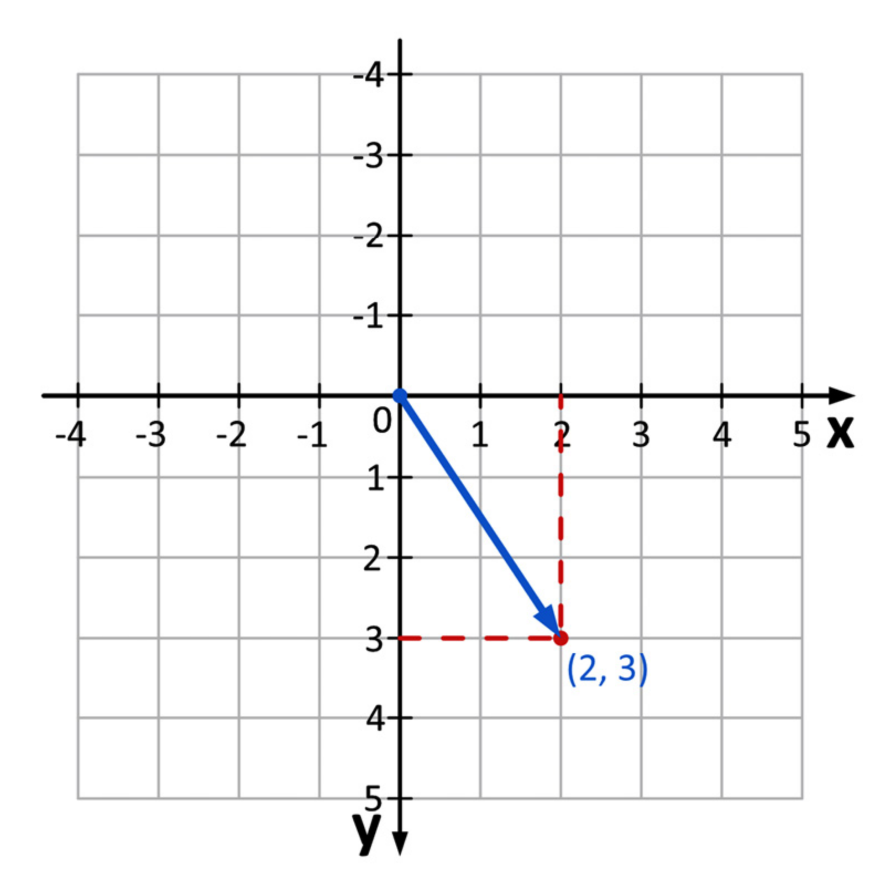


Рисунок 7.3 – Вектор, который определен от (0, 0) до (2, 3)

Математически векторы определяются величиной **магнитуда (magnitude) —** длина (lenght) и **направлением (direction) —**

угол (angle). Мы будем использовать *смещение от начала координат (offset from the origin)*, поскольку это более интуитивно понятно с точки зрения разработки игр.

Математика, которую мы изучим, будет применяться ко всем видам векторов (vectors), положений (positions), и сил (forces), даже когда мы имеем дело с векторами в более чем двух измерениях. Но пока мы ограничимся двумя измерениями.

В Godot 2D-вектор представлен классами **Vector2** и **Vector2i**. Чтобы создать вектор, вы просто указываете значения x и y в конструкторе:

```
var vector: Vector2 = Vector2(2, 3)
```

Теперь мы можем просто получить доступ к переменным x и y этого вектора:

```
vector.x = 2
if vector.y > 0:
   # Остальной код
```

Класс **Vector2** по сути просто содержит компоненты x и y для нашего вектора и, таким образом, может представлять как вектор , так и позицию, как мы видели в предыдущем разделе *Система 2D-координат*.

Vector2i очень похож на **Vector2**, единственное отличие в том, что координаты x и y должны быть целыми значениями (integer), тогда как **Vector2** использует числа с плавающей точкой (floating point numbers). Мы будем придерживаться класса **Vector2** до конца главы и книги.

Масштабирование векторов

Чтобы масштабировать (scale) вектор, делая его длиннее или короче, нам просто нужно умножить или разделить значения x и y на одно и то же число. Число, на которое мы умножаем вектор, называется **скаляр (scalar)**. Например, если мы хотим

сделать вектор в два раза длиннее, мы просто умножаем его значения x и y на **2**:

(x, y) * 2 = (x * 2, y * 2)

На *Рисунке 7.4* показан масштабированный вектор:

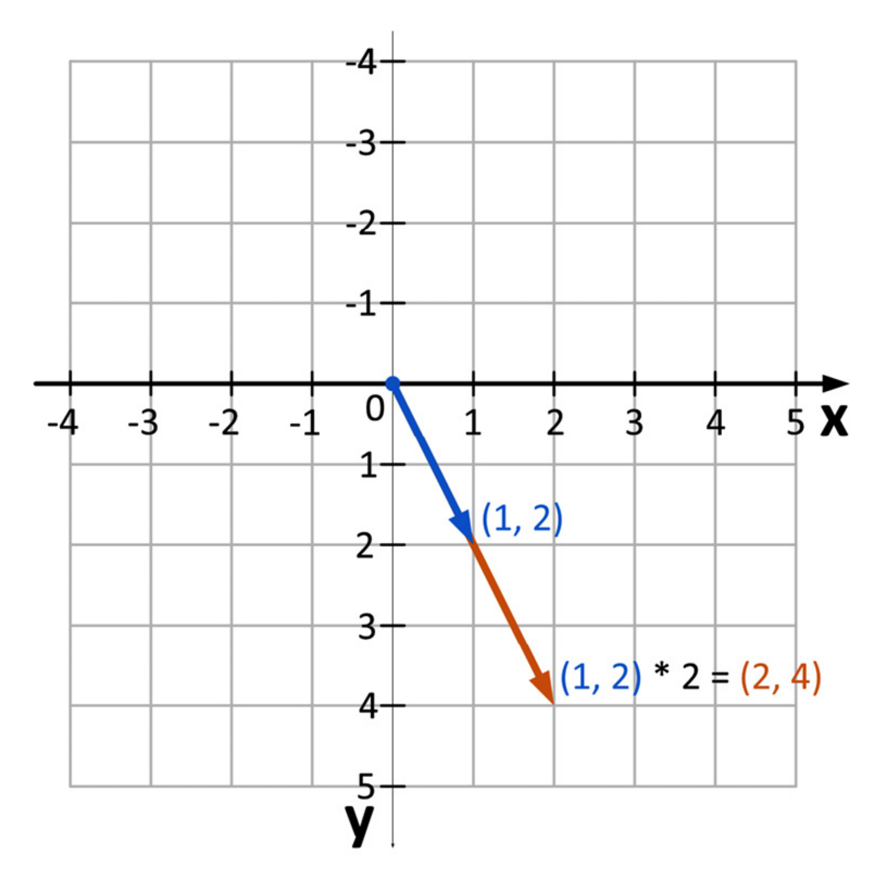


Рисунок 7.4 – Вектор (1, 2), масштабированный на 2, даёт вектор (2, 4)

В Godot мы можем напрямую умножать или делить векторы, используя обычные математические операции, и нам не нужно вручную задавать значения x или y:

```
vector * 2
```

```
vector / 2
```

Интерпретатор выполнит для нас умножение или деление компонентов x и y и выдаст новый вектор с результатом.

Сложение и вычитание векторов

Кроме того, мы можем сложить два вектора. Мы делаем это, складывая значения *x* и принимая результат в качестве значения x для нового вектора, и делаем то же самое для значений *y*:

(x1, y1) + (x2, y2) = (x1 + x2, y1 + y2)

На *Рисунке 7.5* показан результат сложения векторов:

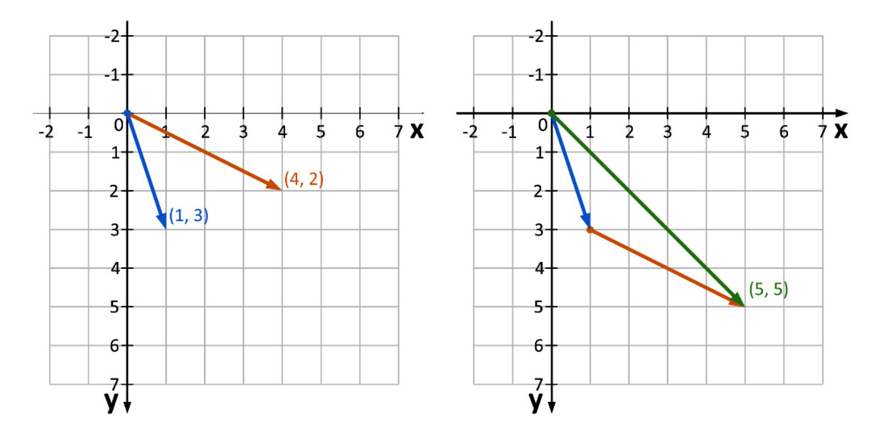


Рисунок 7.5 – Добавление (1, 3) к (4, 2) даёт вектор (5, 5)

Эту операцию можно представить как перемещение сначала в позицию первого вектора , а оттуда — вдоль второго вектора. Это как если бы вы приклеивали второй вектор к концу первого.

Неважно, в каком порядке вы складываете два вектора, результатом операции всегда будет один и тот же вектор.

На *Рисунке 7.6* показана операция в другом порядке:

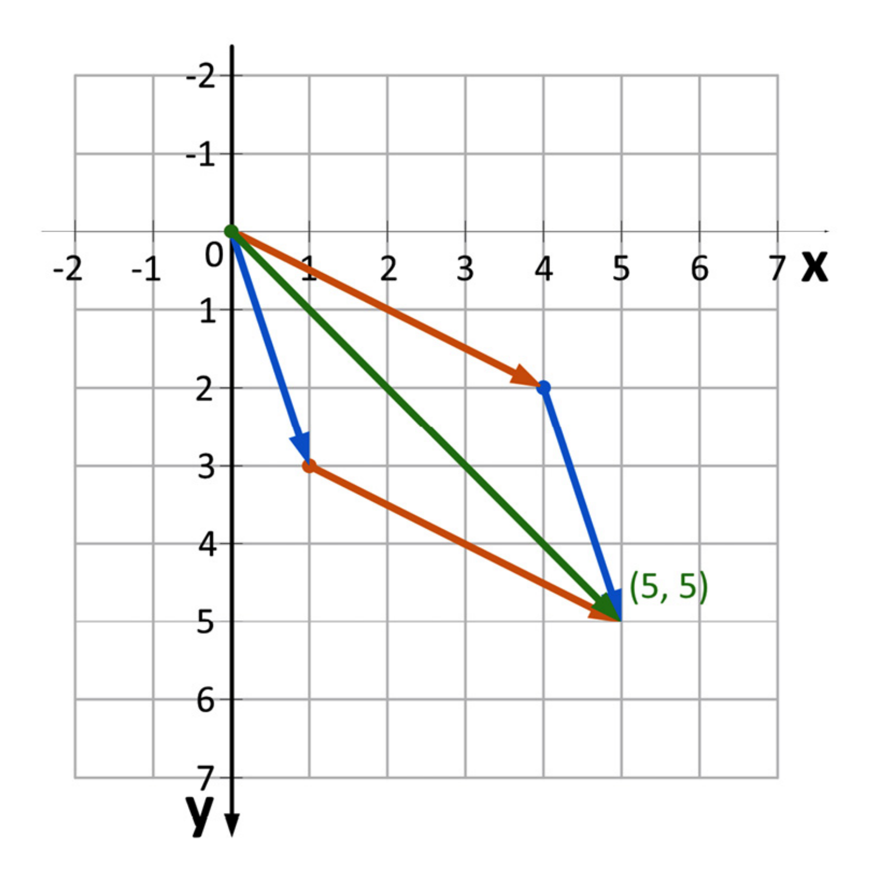


Рисунок 7.6 – (1, 3) + (4, 2) даёт тот же вектор, что и (4, 2) + (1, 3)

Вычитание векторов работает так же, как и их сложение. Просто вычтите отдельные компоненты. **(x1, y1) - (x2, y2)** то же самое, что и **(x1, y1) + (-x2, -y2)**.

В Godot вы можете легко складывать или вычитать векторы, используя обычные операторы плюс и минус:

```
var vector_a: Vector2 = Vector2(2, 3)
var vector_b: Vector2 = Vector2(4, 1)
var vector_c: Vector2 = vector_a + vector_b
var vector_d: Vector2 = vector_a - vector_b
```

Вы увидите, что **vector_c** будет содержать сумму векторов a и b,

а **vector_d** — их разность.

Другие векторные операции

Как вам скажет любой математик, существует гораздо больше формул и операций, которые можно выполнять над векторами. Многие из них очень интересны для нас как разработчиков игр. Однако их немного сложнее объяснить, но большинство из них уже реализовано в Godot. Нам не придется беспокоиться о том, как они работают, хотя знать, что они существуют, очень важно.

Дополнительная информация

Существует гораздо больше замечательных функций, чем мы здесь рассмотрим. Вы можете самостоятельно ознакомиться с документацией по **Vector2**: https://docs.godotengine.org/en/stable/classes/class_vector2.html.

Длина вектора

Длина вектора — это расстояние от начала координат (**0, 0**) до позиции x, y которую представляет вектор. Мы также называем эту длину величиной вектора. Чтобы получить эту длину, мы просто используем функцию **length()**:

```
var vector_length: float = vector.length()
```

Figure 7.7 демонстрирует эту функцию:

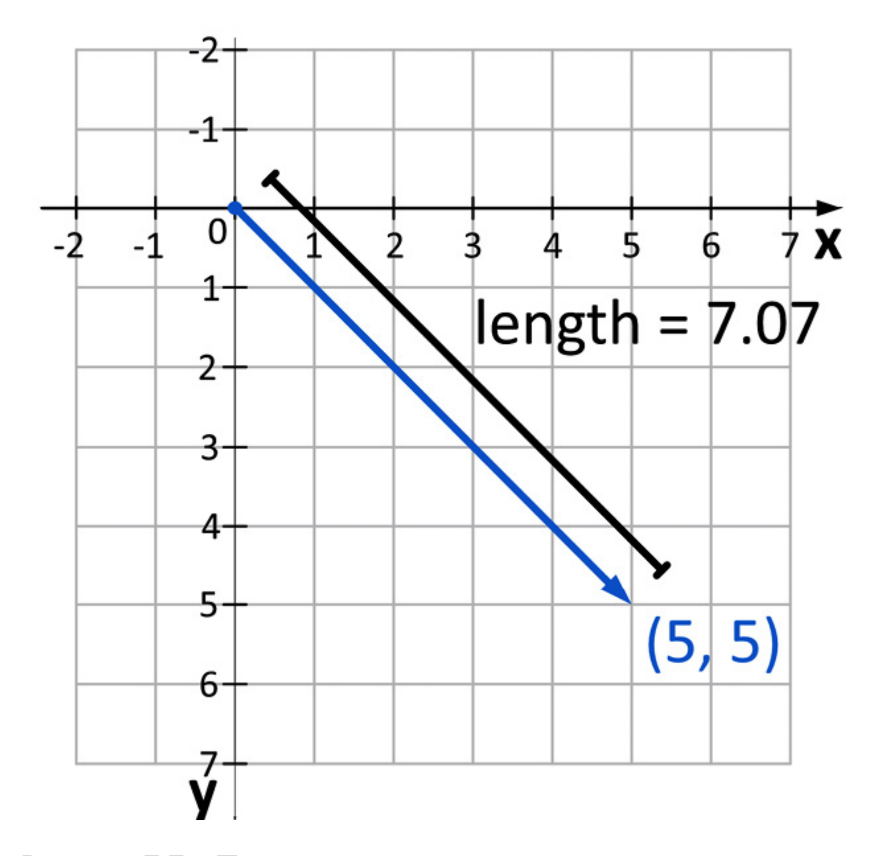


Рисунок 7.7 – Получение длины вектора

На заднем плане функция **length()** использует теорему Пифагора, которая, если бы мы записали её вручную, выглядела бы так:

sqrt(x * x + y * y)

Функция **length()** выполнит все необходимые вычисления за нас и будет работать даже быстрее, чем если бы мы попытались написать эту математику самостоятельно в GDScript.

Ограничение вектора

Иногда вам нужно, чтобы длина вектора оставалась между нижней и верхней границей. Для этого мы используем функцию **clamp()**:

```
var clamped_vector: Vector2 = vector.clamp(0, 5)
```

Рисунок 7.8 демонстрирует эту функцию:

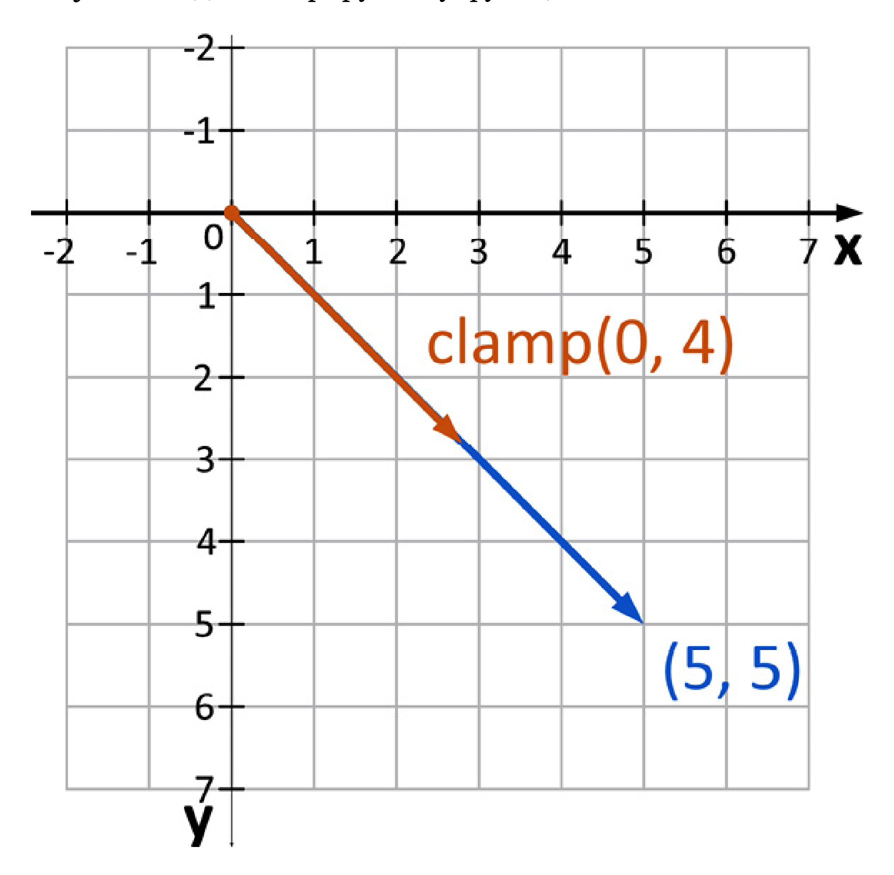


Рисунок 7.8 – Фиксация вектора длиной от 0 до 5

Какой бы ни была длина вектора до этого, после выполнения этого оператора с помощью **clamp()** его длина будет находиться в диапазоне от **0** до **5**.

Расстояние между двумя точками

Часто нам нужно знать расстояние между двумя точками в пространстве. Например, когда мы хотим узнать расстояние между врагом и игроком, чтобы увидеть, находится ли враг в

пределах досягаемости и может ли он выстрелить в игрока. Для этого мы используем функцию **distance_to()**:

```
var distance: float = vector_a.distance_to(vector_b)
```

Figure 7.9 демонстрирует эту функцию:

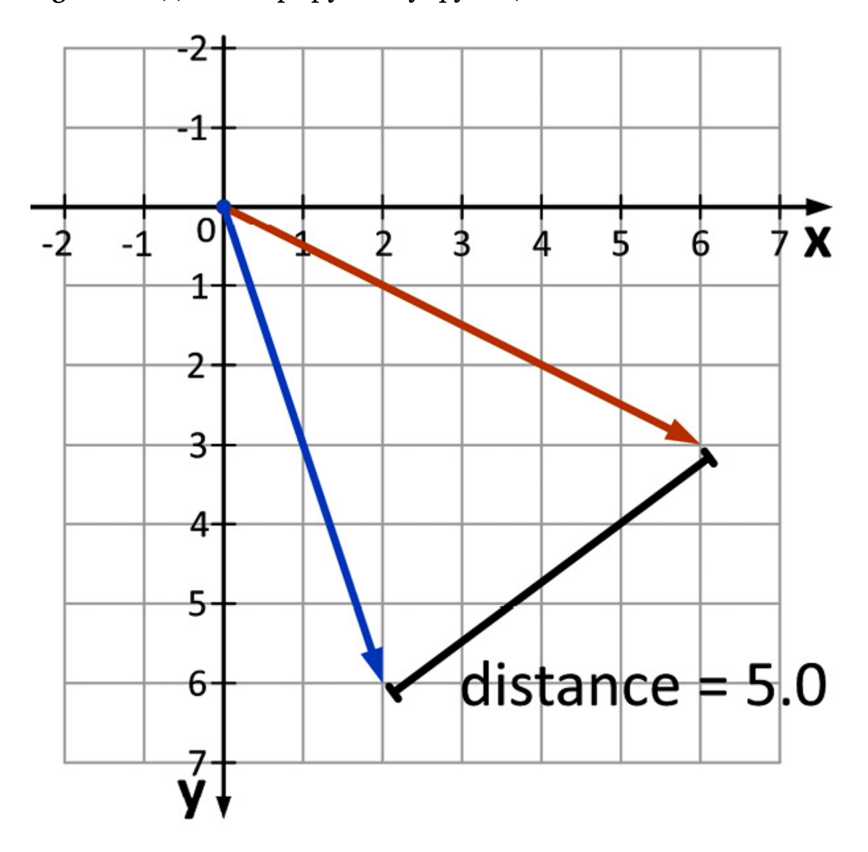


Рисунок 7.9 – Расстояние между двумя векторами

Функция **distance_to()** вернёт число с плавающей точкой, сообщающее нам точное расстояние между двумя точками в пространстве.

Вращение вектора

Вращение вектора или положения полезно при управлении

транспортным средством, вращении игрока или корректировке направления вектора:

```
var vector_b: Vector2 = vector_a.rotated(PI)
```

На *Рисунке 7.10* показана функция **rotated()**:

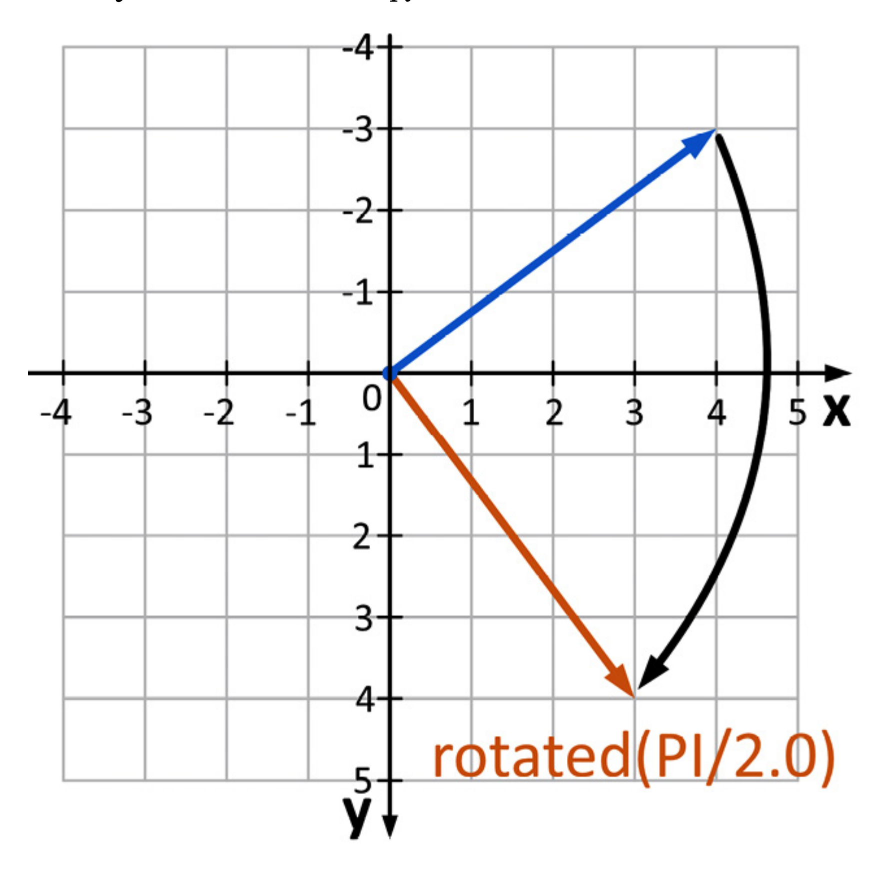


Рисунок 7.10 – Вращение вектора с помощью функции rotated()

Обратите внимание, что по умолчанию углы должны быть в радианах, чтобы Godot мог с ними работать. Если вычисления в радианах — не ваш конек, есть также функция **deg_to_rad()**, которая принимает значение в градусах и преобразует его в радианы:

```
var vector_b: Vector2 = vector_a.rotated(deg_to_rad(180)
```

Функция **deg_to_rad()**, которая принимает значение в градусах и преобразует его в радианы:

Важное примечание

Помните, что полный круг составляет 360 градусов или Pi * 2, также известный как Tau радиан. Это означает, что 1 радиан примерно равен 57,3 градуса.

Нормализация вектора

Нормализация вектора означает, что мы изменяем его длину так, чтобы она была ровно 1. Вектор, который длиннее 1, укорачивается, а вектор, который короче 1, удлиняется. Мы называем полученный вектор **единичным вектором (unit vector)**:

```
var unit_vector: Vector2 = vector.normalized()
```

Figure 7.11 демонстрирует эту функцию:

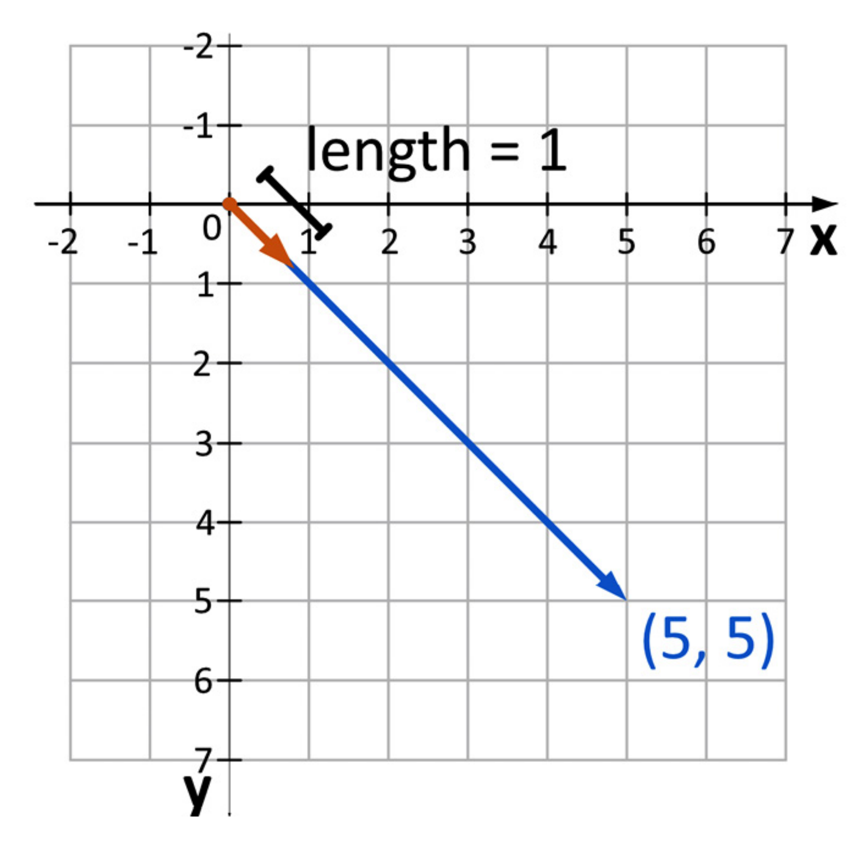


Рисунок 7.11 – Нормализация вектора устанавливает его длину равной 1

Нормализованный вектор полезен для работы, поскольку то, что у нас остаётся, — это направление исходного вектора без его величины. В этой форме мы можем легко использовать единичный вектор для других математических операций, например, когда мы хотим сохранить скорость автомобиля постоянной:

```
velocity = velocity.normalized() * 500.0
```

Таким образом, мы всегда сохраняем скорость — **velocity** на длине 500.0. Углублённое изучение математики, стоящей за всеми этими функциями, наверняка будет полезным! Однако с пониманием, полученным из этого раздела и описанных здесь

функций, мы уже можем сделать много игр. Я рекомендую вам открыть учебник математики или посмотреть обучающие видео в Интернете, но не расстраивайтесь, если вы пока не готовы к этому.

Ресурсы для изучения

Вот три замечательных ресурса, которые помогут вам больше узнать о математике для разработки игр:

В официальной документации Godot есть замечательная статья о векторной математике в движке: https://docs.godotengine.org/en/stable/tutorials/math/vector_math.html.

Книга Джеймса М. Ван Верта и Ларса М. Бишопа *«Основы математики для игр и интерактивных приложений» (Essential Mathematics for Games and Interactive Applications)*: https://www.sciencedirect.com/book/9780123742971/essential-mathematics-for-games-and-interactive-applications.

Серия видеороликов Фрейи Холмер *«Математика для разработчиков игр» (Math for Game Devs)*: https://youtube.com/playlist?list=PLImQaTpSAdsArRFFj8bIfqMk2X7Vlf3XF&si=JZNObwqQNg3ZbGLy.

Итак, освежив в памяти теорию векторной математики , давайте погрузимся в процесс создания движений персонажа игрока.

Перемещение персонажа игрока

После этого отступления к векторной математике, давайте продолжим работу над нашей игрой. Первое, с чего мы начнем, это движение игрока.

Изменение текущего узла игрока

В Godot встроен физический движок. Чтобы его использовать, нам придется использовать физические узлы, предоставляемые самим Godot. Игрок в настоящее время является **Node2D**, но на самом деле мы хотим, чтобы он был **CharacterBody2D**.

К счастью, изменить тип узла очень просто. Для этого выполните следующие действия:

1. Щёлкните правой кнопкой мыши узел **Player**.
2. Выберите **Изменить тип… (Change Type…)**, как показано на следующем рисунке:

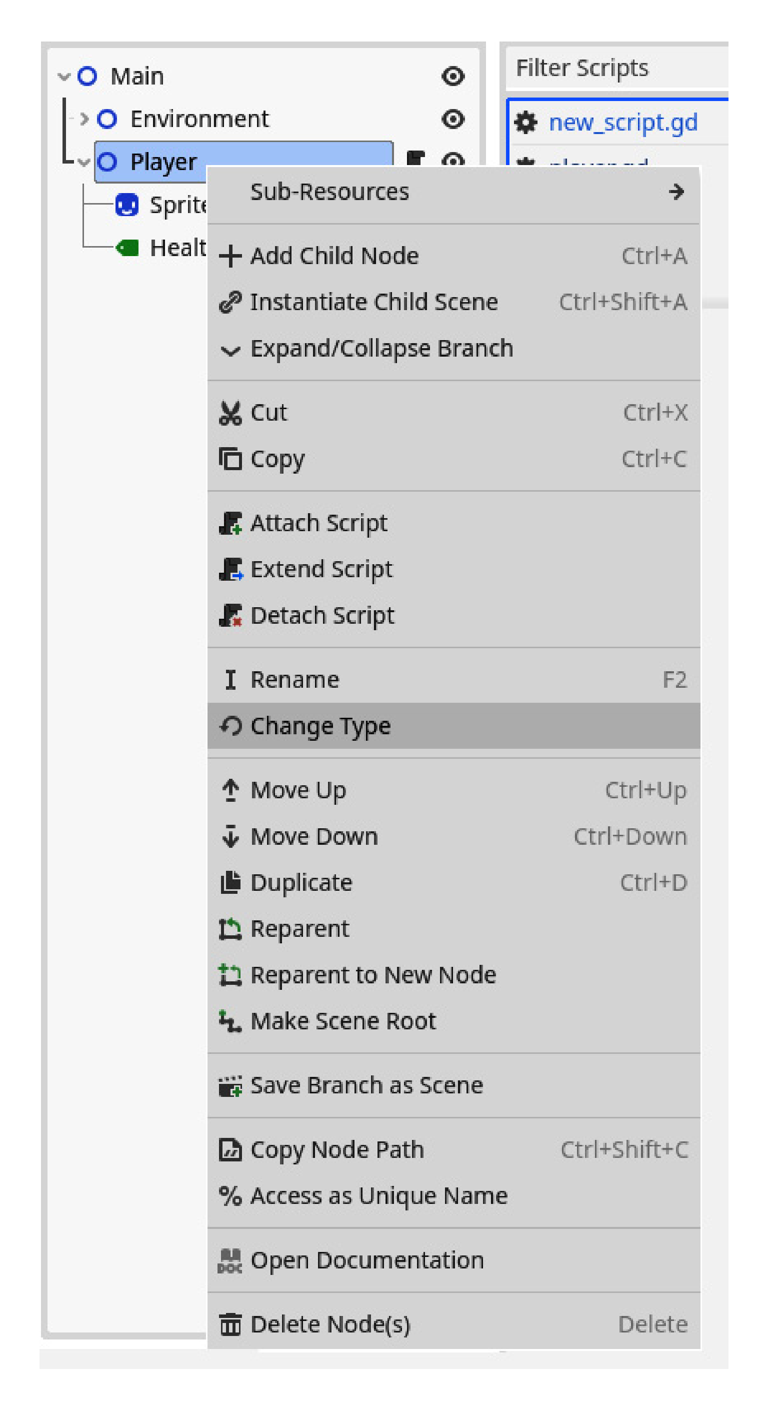


Рисунок 7.12 – Тип узла можно изменить с помощью меню,

которое появляется при щелчке правой кнопкой мыши по этому узлу

1. Найдите **CharacterBody2D** следующим образом:

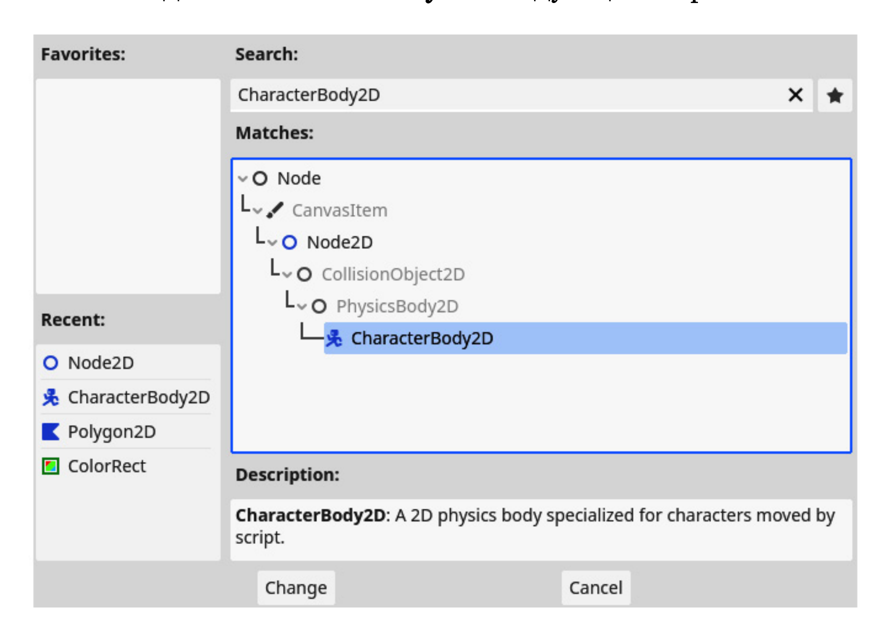


Рисунок 7.13 – Поиск типа узла CharacterBody2D

1. Выберите узел **CharacterBody2D**, и вы увидите, как значок узла **Player** изменится, отражая наш выбор:



Рисунок 7.14 – Узел игрока меняется с Node2D на CharacterBody2D

1. Вы заметите, что появился желтый предупреждающий знак. Если навести на него курсор, то увидите, что он сообщает нам, что игроку нужен **CollisionShape2D** или **CollisionPolygon2D**.

Эти формы помогают нам правильно обрабатывать столкновения. Давайте проигнорируем это пока. Мы займемся обнаружением столкновений в *Главе 9*.

Скрипт нашего игрока будет работать по-прежнему, поскольку узел **CharacterBody2D** является дочерним классом класса **Node2D** и, как мы видели в *Главе 4*, обработка **CharacterBody2D** работает как в **Node2D** из-за принципов полиморфизма.

Мы хотим использовать определённые функциональные возможности узла **CharacterBody2D** в нашем скрипте, поэтому нам придется изменить класс, от которого расширяется наш скрипт игрока, на **CharacterBody2D**:

```
extends CharacterBody2D
# Остальная часть скрипта игрока…
```

Узел **Player** правильно преобразован в физическую сущность. Мы узнаем, как его перемещать в следующем разделе.

Приложение сил к игроку

Теперь, когда персонаж игрока представляет собой физическое тело, мы сможем применять к нему физические силы и заставлять его двигаться!

Прежде чем реализовать полную схему движения, давайте сначала сделаем так, чтобы персонаж двигался вправо. Просто добавьте этот небольшой фрагмент кода в скрипт игрока:

```
func _physics_process(delta: float):
   velocity = Vector2(500, 0)
   move_and_slide()
```

Перемещение узла **CharacterBody2D** необходимо выполнить в два этапа:

1. Рассчитайте и установите переменную скорости — **velocity** для тела. Эта скорость является переменной-членом **CharacterBody2D** и представляет скорость и направление, в котором движется тело. В этом примере я установил её на вектор **(500, 0)**, что означает силу, направленную вправо.
2. Вызовите функцию **move_and_slide()**. Эта функция применит скорость, которую мы только что установили, и выполнит для нас все обнаружение столкновений. Если столкновение обнаружено, эта функция также будет скользить по поверхности этой формы столкновения. Это приводит к естественному движению, которое не слишком за что-то цепляется. *Рисунок 7.15* демонстрирует эту функцию:

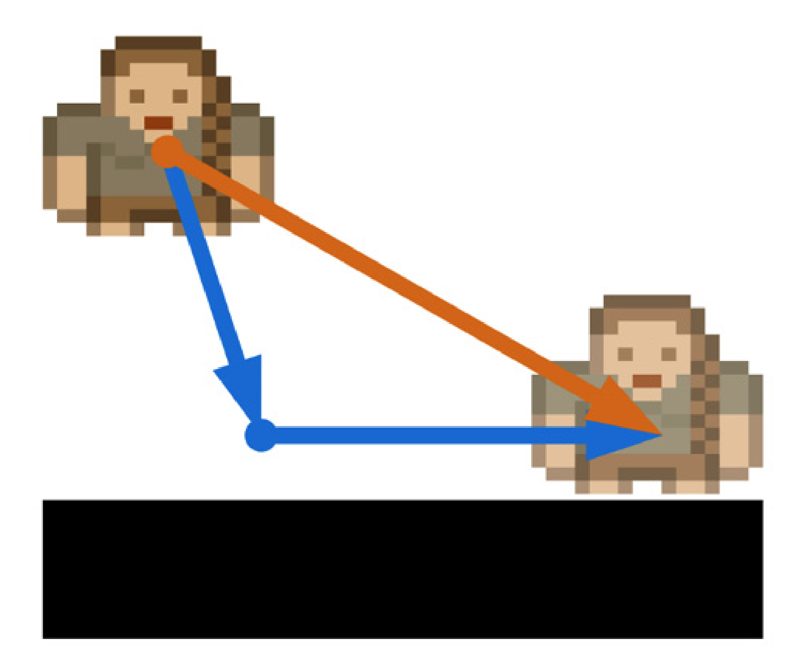


Рисунок 7.15 – move_and_slide() столкновение и скольжение по поверхности

Скорость (velocity) задаёт силу, которую мы хотим применить к персонажу. Последующий вызов **move_and_slide()** действительно применяет эту силу и вычисляет все последствия и взаимодействия между другими физическими телами. На данный момент нет других тел для взаимодействия; мы реализуем их в *Главе 9*.

1. Запустите игру, и вы увидите, как персонаж игрока медленно перемещается по экрану вправо.

В качестве эксперимента попробуйте заставить персонажа двигаться вверх.

В предыдущем фрагменте кода вы можете видеть, что мы использовали функцию **_physics_process()** для выполнения наших расчетов движения. Давайте рассмотрим, почему мы используем именно эту функцию.

Функции _process и _physics process

Мы уже рассматривали функцию **_process()** вкратце в *Главе 1*, но не использовали её в тот раз. Эта функция выполняется для каждого кадра, в котором игра выполняется на каждом узле, что означает, что если игра выполняется 120 кадров в секунду, эта функция выполняется 120 раз в секунду. В реальности частота кадров игры сильно варьируется, поэтому функция **_process()** может быть запущена больше или меньше раз в секунду.

Для физических симуляций изменение частоты кадров является проблемой. Физические расчеты могут быстро стать очень неточными, пропускать столкновения и вносить дрожание, если частота кадров, с которой они выполняются, нестабильна. Вот почему физические движки вводят концепцию физических кадров. Они выполняются со стабильным интервалом и изо всех сил стараются не колебаться. Таким образом, функция **_process()** выполняется столько раз в секунду, сколько возможно, в то время как функция **_physics_process()** выполняется со скоростью, которая является максимально стабильной.

Вот почему мы использовали **_physics_process()** в предыдущем примере. Эта функция вызывается в каждом физическом кадре, каждый раз, когда происходят физические вычисления. По умолчанию в Godot эта частота составляет 60 раз в секунду.

Параметр **delta** предоставляет нам прошедшее время с момента последнего вызова функции **_physics_process()**, как и параметр **delta** в обычной функции**_process()**. Когда у нас есть 60 физических кадров в секунду, это время должно быть стабильным на уровне 1,0 / 60,0 кадров в секунду = 0,01667 секунды.

Обе функции, **_process()** и **_physics_process()**, работают очень похоже — они периодически вызываются во время выполнения игры. Но для физических расчетов мы будем использовать **_physics_process()**.

Список действий

Чтобы переместить персонажа игрока в направлении, которое задумал игрок, нам сначала нужно получить ввод от игрока. Этот ввод обычно осуществляется через клавиатуру, мышь или контроллер.

Общие методы ввода для перемещения персонажа видеоигры приведены в *Таблице 7.1*:

Analog stick
Аналоговый джойстик
Двигаться влево
Move left
Двигаться вправо
Move right
Двигаться вверх
Move up
Двигаться вниз
Move down

Таблица 7.1 – Методы ввода для перемещения персонажа видеоигры

Чтобы согласовать все эти различные входы и устройства в одну согласованную систему, Godot имеет встроенный инструмент **список действий (input map)**. Он будет управлять согласованием, так что мы можем просто сосредоточиться на использовании ввода в игре.

Список действий в основном группирует **события ввода (input events)**, такие как стрелка влево (left arrow), клавиша *A*, движение джойстика влево и левая кнопка на D-pad) в одно **действие (action) —** moving left. Всё, что нам нужно сделать, это определить каждое действие и связать все операции ввода, которые мы хотим поддерживать.

Давайте настроим нашу карту ввода:

1. Откройте **Настройки проекта (Project Settings)**. Вы найдете их в пункте меню **Проект (Project)** в верхней части редактора.
2. Затем перейдите на вкладку **Список действий (Input Map)**:

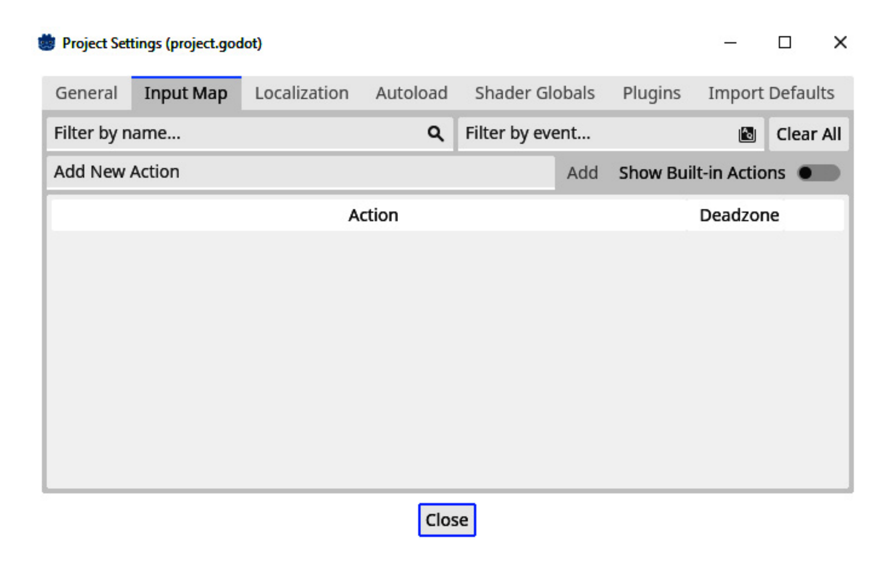


Рисунок 7.16 – Вкладка «Список действий» в настройках проекта

1. Введите **move_left** в поле ввода с надписью **Добавить**

новое действие (Add New Action) и нажмите **Добавить (Add)**. Теперь вы увидите новое действие, появившееся в главном разделе окна:

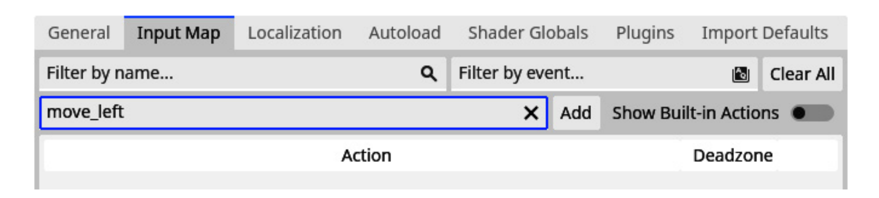


Рисунок 7.17 – Добавление нового действия с именем **move_left**

1. Нажмите знак + рядом с названием действия, чтобы добавить событие к нашему действию.
2. Теперь, выбрав верхнее поле ввода, нажмите клавишу со стрелкой влево на клавиатуре.
3. Убедитесь, что в нижнем раскрывающемся меню выбран пункт **Физический код клавиши (Physical Keycode)** и нажмите **OK**, чтобы добавить событие к действию:

Event Configuration
Left (Physical)
Left or Left (Physical) or Left (Unicode)
Filter Inputs
Left
Up
Right
Down
PageUp
PageDown
Additional Options
Alt Shift Ctrl Windows Command / Control (auto)
Physical Keycode (Position on US QWERTY Keyboard)
OK Cancel

Рисунок 7.18 – Настройка события на клавишу со стрелкой

влево

Обратите внимание, что физический код клавиши выбран внизу.

1. Повторите *шаги с 3 по 5* для клавиши *A*, если вы используете QWERTY-клавиатуру, или используйте эквивалентную клавишу для раскладки вашей клавиатуры.
2. Повторите *шаги с 3 по 5* для движения аналогового джойстика и кнопки D-pad контроллера, если он у вас есть. Если у вас нет контроллера для подключения к компьютеру, вы можете просто поискать следующие события в меню:

   - **Joypad Axis 0** (left stick, joystick 0 left)
   - **Joypad Button 13** (D-pad left)

5. Теперь создайте действия **move_right**, **move_up** и **move_down** и повторите предыдущие шаги, чтобы добавить все соответствующие входные события к каждому из них.

Когда вы закончите, ваш список действий должна выглядеть примерно так:

Action	Deadzone
move_left	0.5
Left (Physical)	
Joypad Axis 0 - (Left Stick Left, Joystick 0 Left) - All Devices	
Joypad Button 13 (D-pad Left) - All Devices	
move_right	0.5
Right (Physical)	
Joypad Axis 0 + (Left Stick Right, Joystick 0 Right) - All Devices	
Joypad Button 14 (D-pad Right) - All Devices	
move_up	0.5
Up (Physical)	
Joypad Axis 1 - (Left Stick Up, Joystick 0 Up) - All Devices	
Joypad Button 11 (D-pad Up) - All Devices	
move_down	0.5
Down (Physical)	
Joypad Axis 1 + (Left Stick Down, Joystick 0 Down) - All Devices	
Joypad Button 12 (D-pad Down) - All Devices	

Рисунок 7.19 – Полное отображение списка действий для перемещения персонажа игрока

Физический код клавиши

Каждая клавиша на клавиатуре имеет уникальный скан-код. Это простое число, которое идентифицирует эту уникальную клавишу. Этот скан-код интерпретируется операционной системой в код клавиши, то есть букву, символ или любую другую операцию на клавиатуре. Эта интерпретация учитывает раскладку клавиатуры (QWERTY, AZERT и т. д.). Проблема в том, что клавиши располагаются в разных местах в зависимости от раскладки клавиатуры и используемой вами операционной системы.

Чтобы упростить это, Godot имеет физические коды клавиш, которые напрямую используют скан-код клавиши вместо ее базового символа или операции. Таким образом, мы гарантируем, что клавиши перемещения (которые на клавиатуре QWERTY являются WASD, но на клавиатуре AZERTY являются ZQSD) всегда находятся в одном и том же месте,

независимо от раскладки клавиатуры ..

Использование ввода

Использовать действия ввода, которые мы настроили для направленного движения персонажа игрока, в Godot довольно просто, поскольку в игре есть встроенная функция, которая может выполнять четыре действия и выдавать вектор, представляющий направление, в котором игрок хочет двигаться.

Эта функция определена на глобальном объекте **Input** и называется **get_vector()**. Она принимает четыре аргумента, которые являются именами действия для отрицательного направления x, положительного направления x, отрицательного направления y и положительного направления y.

Одиночки — Singletons

Глобальные объекты — это объекты, доступные из любой части кодовой базы и автоматически создаваемые экземпляры. Их также называют синглтонами или автозагрузками.

Чтобы заставить нашего персонажа двигаться, нам нужно сделать следующее:

```
func _physics_process(delta: float):
   var input_direction: Vector2 = Input.get_vector("move
   velocity = input_direction * 500.0
   move_and_slide()
```

Вы можете видеть, что мы сохраняем направление ввода в переменной **input_direction** , а затем умножаем этот вектор направления на 500. Это происходит потому, что вектор, возвращаемый **get_vector()** , имеет длину 1, что очень мало и делает движение нашего персонажа практически неотличимым от стояния на месте.

При запуске игры мы наконец-то можем перемещать нашего персонажа, используя все различные методы ввода, которые мы

настроили.

В качестве эксперимента попробуйте не умножать направление ввода на 1000.

Сглаживание движения

Теперь движение игрока уже работает довольно хорошо, но кажется очень жёстким. С того момента, как наш персонаж начинает двигаться, он движется с максимальной скоростью, а когда мы отпускаем все кнопки, персонаж мгновенно останавливается. Так вещи не движутся в реальном мире. Всегда есть период ускорения и замедления до и после движения.

Давайте сначала определим некоторые экспортные переменные, чтобы мы могли поиграться со всеми различными частями движения. Это значительно облегчит настройку и полировку.

В верхней части скрипта player.gd добавьте переменную экспорта для максимальной скорости, ускорения и замедления:

```
@export var max_speed: float = 500.0       # максимальна
@export var acceleration: float = 2500.0   # ускорение
@export var deceleration: float = 1500.0   # замедление
```

Затем перепишите функцию **_physics_process()** следующим образом:

```
func _physics_process(delta: float):
   var input_direction: Vector2 = Input.get_vector("move
   if input_direction != Vector2.ZERO:
      velocity = velocity.move_toward(input_direction *
   else:
      velocity = velocity.move_toward(Vector2.ZERO, dece
   move_and_slide()
```

Vector2.ZERO — это просто хорошая замена **Vector2(0, 0)**. Она

не короче, но её легче читать.

You can see that we put the input vector in a new variable. Then we checked whether it was equal to the zero vector **(0, 0)**. В этом случае игрок вводит данные, поэтому нам нужно ускориться в направлении **input_direction**. Мы делаем это с помощью функции **move_toward()**. Эта функция переместит вектор, который мы ей вызываем, который в данном случае является переменной **velocity**, чтобы он совпал с другим вектором, который в данном случае является **input_direction** масштабированным на **max_speed**. Масштабированное **input_direction** является первым параметром для расстояния, на которое мы движемся, что является вторым параметром. Самое замечательное в **move_toward()** то, что он не превышает целевой вектор, поэтому нам не нужно выполнять никаких дополнительных вычислений. *Рисунок 7.20* демонстрирует это:

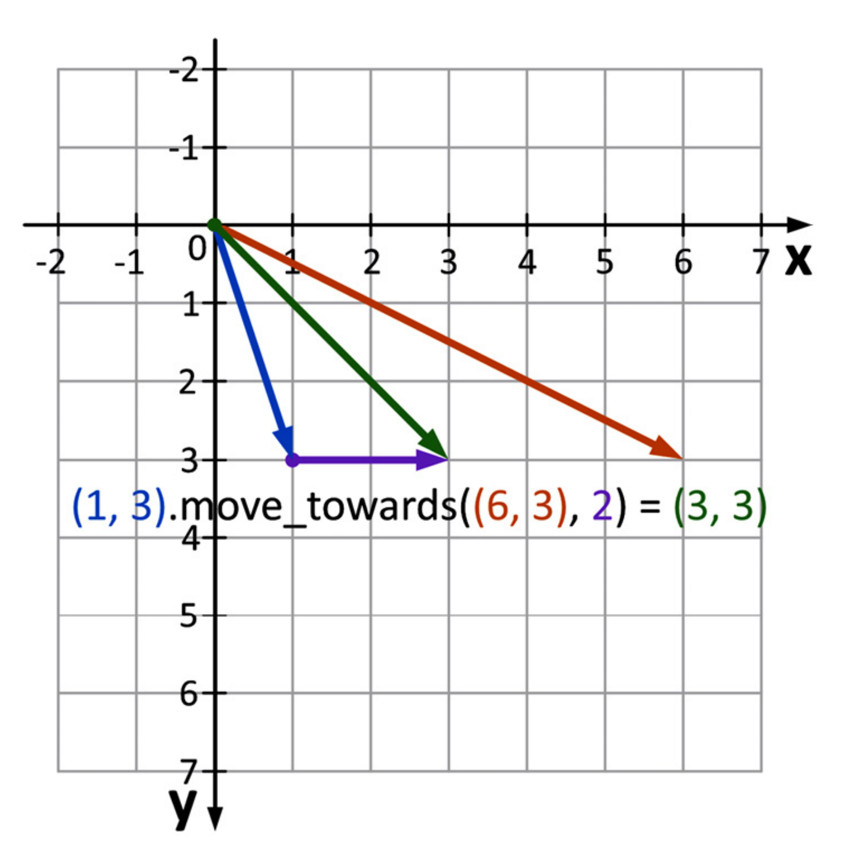


Рисунок 7.20 – move_towards демонстрирует, что перемещение от (1, 3) к (6, 3) на 2 единицы приводит к (3, 3)

Вектор, к которому мы хотим двигаться, — это **input_direction**, умноженное на **max_speed**, что по сути является вектором, который мы имели в предыдущем разделе, *Использование ввода*.

Величина, с которой мы пытаемся двигаться к новому вектору, это ускорение — **acceleration**, умноженное на **delta**, которая входит в функцию **_physics_process()**. Это гарантирует, что ускорение и замедление всегда применяются с одинаковой скоростью и не зависят от частоты кадров. Нам не нужно умножать **velocity** на **delta**, поскольку **move_and_slide()** уже включает **delta** между физическими кадрами по умолчанию.

В случае, когда **input_direction** равен нулевому вектору, мы хотим замедлиться. Мы делаем это точно так же, как и ускоряемся, перемещая скорость к нулевому вектору, используя замедление — **deceleration**, как величину этого движения.

Когда вы запустите игру сейчас, вы увидите, что движение ощущается намного лучше. Вы можете немного поиграть с экспортированными переменными, чтобы настроить все по своему вкусу.

Важное примечание

Помните, что вы изменяете эти переменные во время работы игры, поскольку мы экспортировали переменные!

В этом разделе мы узнали, как настроить физический объект для перемещения и как обрабатывать направленный ввод, а также более глубоко погрузились в плавное движение. Далее давайте узнаем, как отлаживать игру во время её работы.

Отладка запущенной игры

До сих пор мы отлаживали нашу игру, выводя значения на консоль вывода. Это очень быстрый и эффективный способ отладки, но существуют опции, которые могут пролить гораздо

больше ясности на то, что происходит во время выполнения нашей игры.

Давайте рассмотрим эти способы подробнее.

Точки останова — Breakpoints

Самый лассический способ отладки — использование **точек останова (breakpoints)**. Точка останова буквально прерывает или останавливает программу на строке кода, в которой она была установлена. Чтобы установить точку останова, щёлкните рядом с номером строки кода в редакторе кода.

```
func _physics_process(delta: float):
	var input_direction: Vector2 = Input

	if input_direction != Vector2.ZERO:
		velocity = velocity.move_toward(
	else:
		velocity = velocity.move_toward(

	move_and_slide()
```

Рисунок 7.21 – Добавление точки останова в код

Когда интерпретатор достигает этой строки, а значит и точки останова, он останавливает всё и показывает, где находится выполнение кода.

Попробуйте поместить точку останова в функцию **_physics_process()** в скрипте player.gd. После остановки программы снизу развернётся панель **Отладчик (Debug)**. На этой панели вы можете изучить кучу разных вещей:

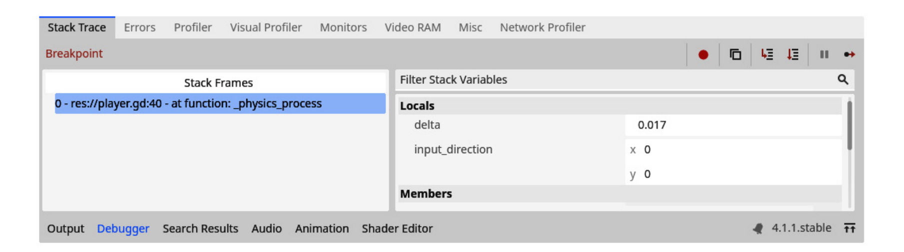


Рисунок 7.22 – Панель отладчика показывает контекст точки останова

Справа вы можете увидеть все переменные, которые в данный момент находятся в области действия вместе с их значениями. Это невероятно полезно, поскольку мы можем увидеть состояние всех переменных одним быстрым взглядом. С помощью оператора print() вы могли видеть только те значения, которые мы ввели в оператор печати, но теперь мы видим все текущие значения в области действия:

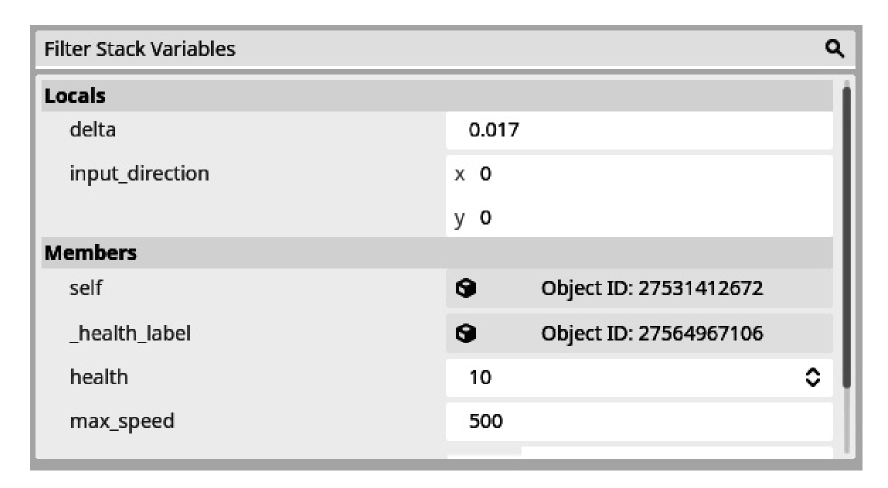


Рисунок 7.23 – Все значения переменных, которые в данный момент находятся в области видимости, отображаются на панели отладчика

С левой стороны панели вы можете видеть текущий стек, то есть различные функции, через которые прошла программа, чтобы оказаться здесь. На данный момент это просто функция **_physics_process()**, поскольку это единственная выполняемая функция. Позже эта панель покажет нам последовательность

функций, которые были вызваны, чтобы добраться до этой точки в коде:

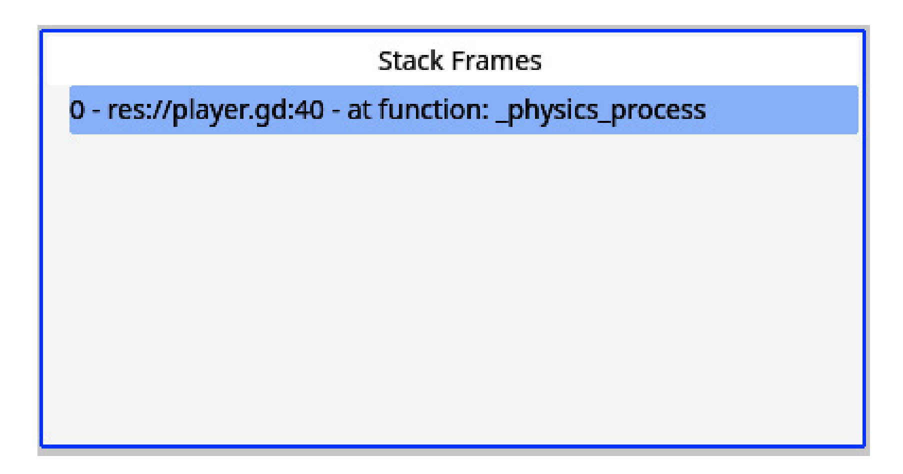


Рисунок 7.24 – Кадр стека показывает все функции, выполненные для достижения точки останова

В правом верхнем углу панели расположены некоторые элементы управления отладкой:

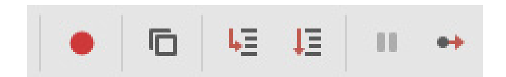


Рисунок 7.25 – Все элементы управления отладкой

Эти элементы управления позволяют нам решать, что делать дальше, когда мы достигли точки останова:

- **Пропускать точки останова (Skip breakpoints)**: этот параметр определяет, должен ли интерпретатор останавливаться на точках останова или нет.
- **Копировать ошибку (Copy error)**: копирует текст текущей ошибки в буфер обмена, готовый к вставке в поиск Google. Это не очень применимо сейчас, но очень удобно при столкновении с ошибкой.
- **Шаг вглубь (Step into)**: это заставит интерпретатор сделать один шаг в коде и войти во все определённые пользователем функции, которые он встретит.
- **Шаг с обходом (Step over)**: то же самое, что и **Шаг**

вглубь (Step Into), но не переходит в функции.
- **Прерывание (Break)**: эта кнопка остановит игру, независимо от того, где в данный момент происходит выполнение.
- **Продолжить (Continue)**: Это позволит продолжить обычное выполнение игры.

Эти кнопки помогут нам перемещаться по коду, находясь в точке останова.

Важное примечание

Точки останова существуют практически во всех языках программирования, таких как JavaScript и Python, и редакторах кода, таких как Jetbrains' Rider, в той или иной форме. Ознакомиться с их работой весьма полезно.

Удалённое дерево — Remote tree

Второй способ отладки специфичен для Godot. Поскольку каждая сцена строится из дерева сцены, а это дерево конструируется и напрямую изменяется во время выполнения игры, было бы полезно взглянуть на то, как выглядит дерево и его узлы в каждой точке игры. И вы правы, именно это и есть **удалённое дерево (remote tree)**!

Сначала запустите игру. В редакторе прямо над деревом сцен появятся две новые кнопки: **Удалённый (Remote)** и **Локальный (Local)**. В данный момент мы находимся в **Локальном** дереве сцены, которое является деревом сцены, которую мы редактируем:

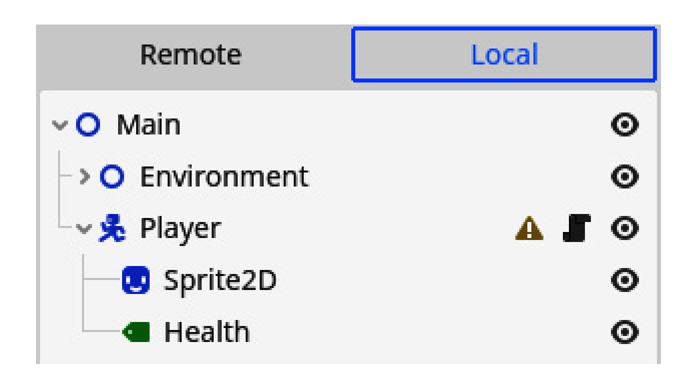


Рисунок 7.26 – Локальное дерево сцены во время выполнения нашей игры

Если мы переключимся на **Удалённый**, мы переключимся на дерево, каково оно есть в запущенной игре:

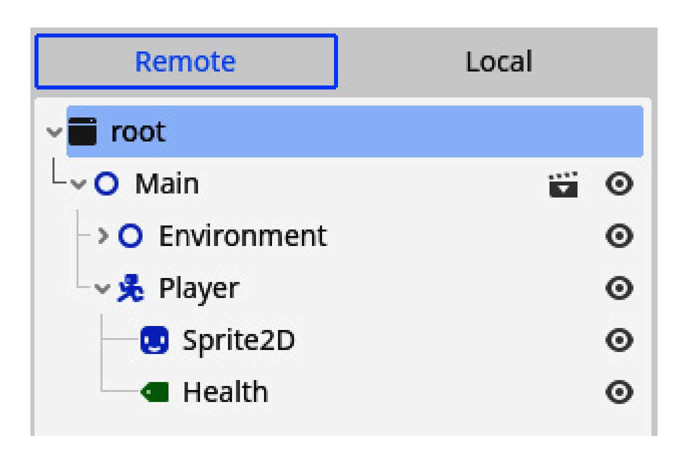


Рисунок 7.27 – Переключение на удалённое дерево с помощью кнопки Удалённый

Вы можете выполнить поиск по удалённому дереву для узла **Player**. Если вы щёлкните по нему, инспектор покажет вам значения этого узла, как они есть в игре. Если вы прокрутите вниз до положения узла и немного переместите игрока в игре, вы увидите, что значения в инспекторе изменяются в соответствии с положением персонажа игрока:

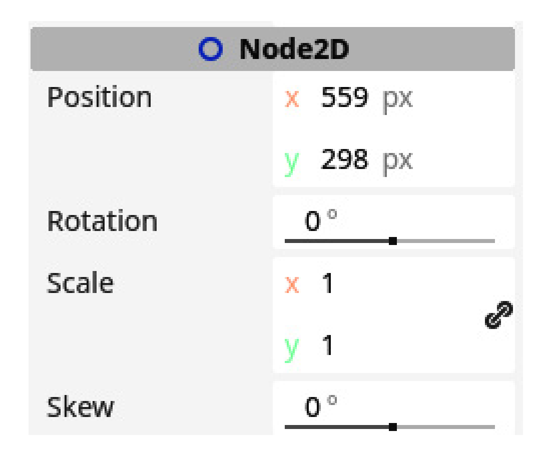


Рисунок 7.28 – Изменение параметра Position узла в удаленном дереве

Если вы не понимаете, что происходит, вспомните функцию print, точки останова и удалённое дерево, и вы наверняка поймете, в чем ошибка!

Дополнительные упражнения – Заточка топора

Вот еще несколько упражнений, которые вы можете выполнить в Godot:

1. Измените параметры движения персонажа игрока так, чтобы создавалось ощущение катания на коньках.
2. Перепишите функцию **_physics_process()** в скрипте player.gd, заменив функцию **move_toward()** на необработанные векторные операции (raw vector operations). Вам нужно будет добавить вектор к переменной **velocity** в направлении **input_direction** , который имеет длину переменной ускорения — **acceleration**. Затем вам нужно будет убедиться, что переменная скорости, **velocity**, не превышает значение переменной **max_speed**.

Итоги

Эта глава была полностью посвящена физике и перемещению персонажа игрока. Мы даже немного обновили наши познания в векторной математике. Теперь персонаж игрока может перемещаться за пределы экрана! Более того, он может бегать сквозь стены.

В *Главе 9* мы исправим эти проблемы с помощью обнаружения столкновений и даже создадим предметы коллекционирования, например деньги.

Во-первых, нам нужно будет научиться разделять наше дерево сцены на части — самостоятельные файлы сцен и использовать их. мы совершим небольшое отступление в следующей главе.

Опрос

- Каким классом представлены векторы в GDScript?
- Решите следующие векторные математические уравнения:
    - (2, 4) + (-4, 3)
    - (-1, 2) - (6, 6)
    - (3, 1) * 2
- Если мы хотим переместить игрока с помощью физического движка, нужно это делать в функции **_process()** или в **_physics_process()**?
    - Сколько раз в секунду вызывается функция **_process()**?
    - Сколько раз в секунду вызывается функция **_physics_process()**?
- Почему мы использовали физический код клавиши (Physical Keycode) для регистрации кнопок клавиатуры в списке действий (input map) нашего проекта?

8

Разделение и повторное использование сцен

Можно создать всю игру в рамках одной сцены Godot, но это может стать довольно громоздким. Нам не только придётся воссоздавать каждую часть сызнова, например, каждый валун или врага. Кроме того, если мы захотим что-то изменить в камнях, нам придётся найти каждый камень в сцене, чтобы изменить его.

Это не годится для любого типа игры. К счастью, в Godot есть такие вещи, как *сцены (scenes)*. В *Главе 2* мы увидели, как создавать новые сцены с нуля, но в этой главе мы узнаем, как можно создать отдельную сцену для каждого элемента, чтобы мы могли легко повторно использовать её на протяжении всей игры. Таким образом, мы можем создать одну сцену для камней и использовать её для заполнения арены вместо того, чтобы иметь несколько уникальных камней.

Помимо повторного использования компонентов, гораздо проще работать не над большой сценой, а над определёнными частями игры по отдельности. Сохраняя части игры таким образом, мы можем сосредоточиться на том, над чем работаем.

Другие игровые движки имеют очень похожие системы. В Unity есть префабы, Unreal Engine имеет Blueprint Classes и т.д. Самое замечательное в сценах Godot то, что они ведут себя так же, как и любой другой узел, как только их экземпляры создаются (инстанцируются) в дереве сцены.

В этой главе мы рассмотрим следующие основные темы:

- Сохранение ветки как новой сцены
- Использование сохранённых сцен

- Организация сцен в проекте

Технические требования

Как и для каждой главы, окончательный код можно найти в репозитории GitHub этой книги в подпапке для этой главы: https://github.com/PacktPublishing/Learning-GDScript-by-Developing-a-Game-with-Godot-4/tree/main/chapter08.

Сохранение ветки как новой сцены

В разделе *Создание новых сцен* *Главы 2* мы узнали, как создавать новые сцены для различных экспериментов с кодом. Этот процесс можно использовать для создания любой сцены. Но есть другой способ сделать это — сохранить часть существующего дерева сцены. Мы разделим ветвЬ дерева сцены на отдельнst подсценѕ, которые можно будет повторно использовать где угодно.

Создание отдельной сцены персонажа игрока

Давайте сохраним узел **Player** как отдельную сцену, чтобы мы могли работать над ним изолированно. Перейдите на нашу сцену **Main** и выполните следующие шаги:

1. Щёлкните правой кнопкой мыши узел **Player**.
2. В раскрывающемся меню выберите **Сохранить ветку как сцену (Save Branch as Scene)**:

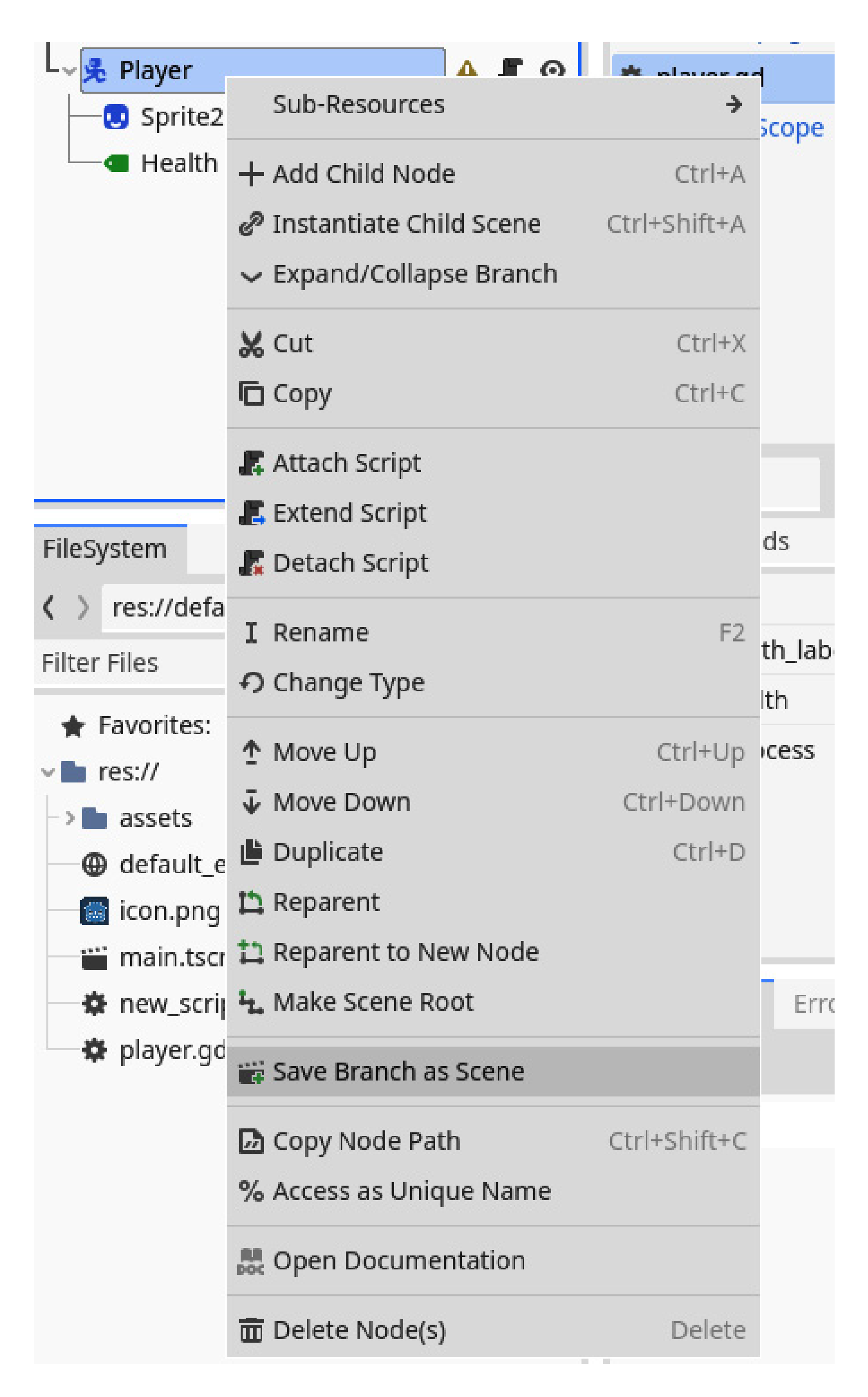


Рисунок 8.1 – Выбор «Сохранить ветку как сцену» для

сохранения узла как отдельной сцены

1. Теперь нам нужно выбрать расположение и название для новой сцены. Оставим всё как есть, это должно сработать:

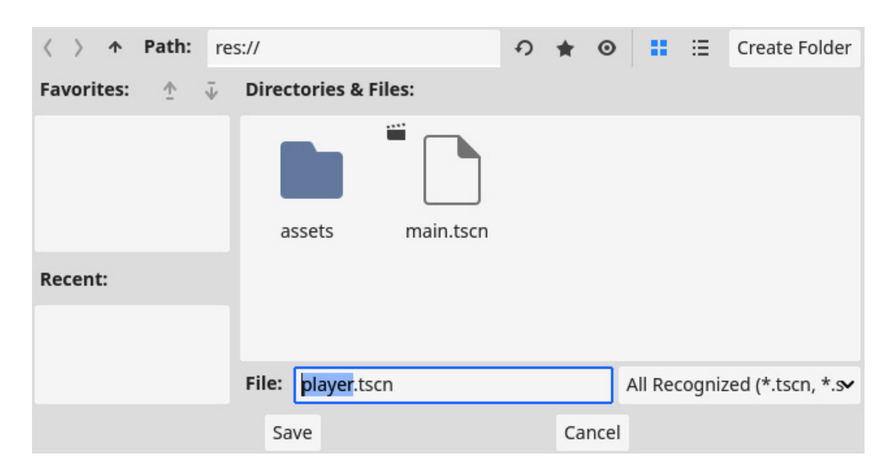


Рисунок 8.2 – Сохранение сцены под соответствующим именем

1. Теперь откроется новая сцена , содержащая только узел **Player** и его дочерние элементы:

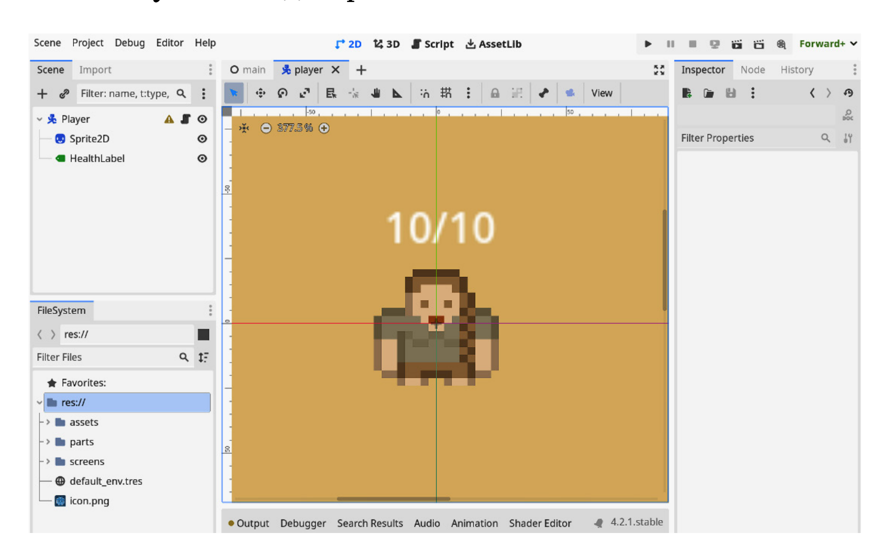


Рисунок 8.3 – Сцена player.tscn, которая содержит только узел Player и его дочерние элементы

1. Сбросьте параметр Position узла **Player** так, чтобы он находился в точке **(0, 0)** внутри сцены:

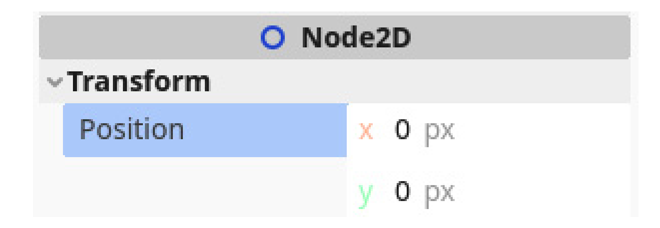


Рисунок 8.4 – Сброс положения корневого узла Player на (0, 0)

Если вернуться к сцене Main, как показано на *Рисунке 8.3*, то можно увидеть, что узел **Player**, который изначально имел несколько дочерних узлов под ним, заменён одним узлом с именем **Player**. Этот узел теперь представляет всё, что находится в сцене **Player**. Визуально в 2D-редакторе ничего не изменилось — игрок всё ещё там полностью, с его меткой **Health** и спрайтом **Sprite2D**:

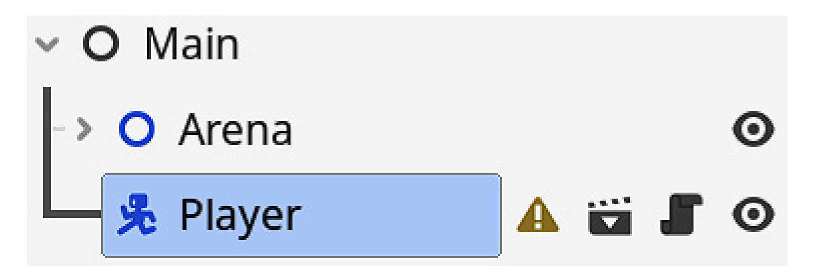


Рисунок 8.5 – Узел Player и его дочерние элементы заменяются одним узлом

Если вы запустите игру сейчас, ничего не изменится , потому что всё осталось прежним. Мы просто отделили узел **Player** в отдельный файл сцены. Вы можете проверить это, перейдя в режим **Удалённый (Remote)** дерева сцены и убедившись, что узел player раскрывает все свои части, когда игра начинает работать:

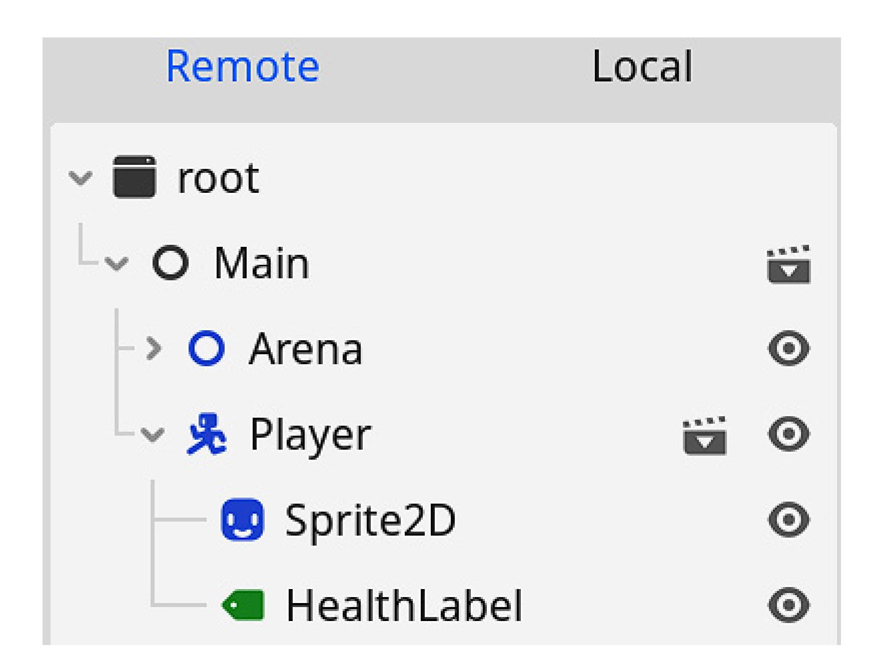


Рисунок 8.6 – Узел Player раскрывается, чтобы все его дочерние узлы были в удалённом дереве при запуске игры

Кроме того, в узле **Player** теперь доступна новая кнопка. Нажатие этой кнопки перенесет нас прямо в сцену **Player**. Это очень удобно при работе со множеством различных сцен и узлов:

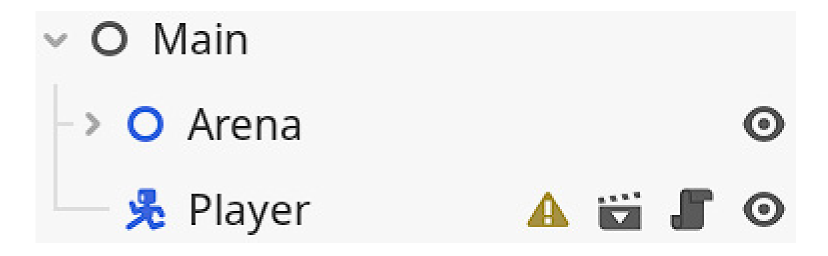


Рисунок 8.7 — Новая кнопка, которая перенесёт нас прямо в сцену Player

Теперь у игрока есть своя собственная сцена, в которой он может работать, не взаимодействуя со всем происходящим в игре.

Корневой узел сцены

Вы также увидите, что корневой узел сцены игрока — это узел **CharacterBody2D** с именем **Player**, который мы выбрали для него в *Главе 7*. Сцены могут иметь любой тип узла в качестве корня. Вы можете выбрать этот тип при создании сцены, как мы сделали в *Главе 2*, или изменив тип узла позднее, как мы сделали это в *Главе 7* для узла **Player**.

Наличие отдельного файла сцены позволяет нам создавать несколько экземпляров этой сцены внутри другой сцены. Мы увидим, как это можно сделать в следующем разделе.

Использование сохранённых сцен

Поскольку мы будем использовать только одного игрока в игре, мы не будем повторно использовать сцену игрока несколько раз. Однако мы повторно используем на арене камни и стены. Следуйте инструкциям в разделе *Сохранение ветви как новой сцены*, чтобы выделить один валун (boulder) в новую сцену:

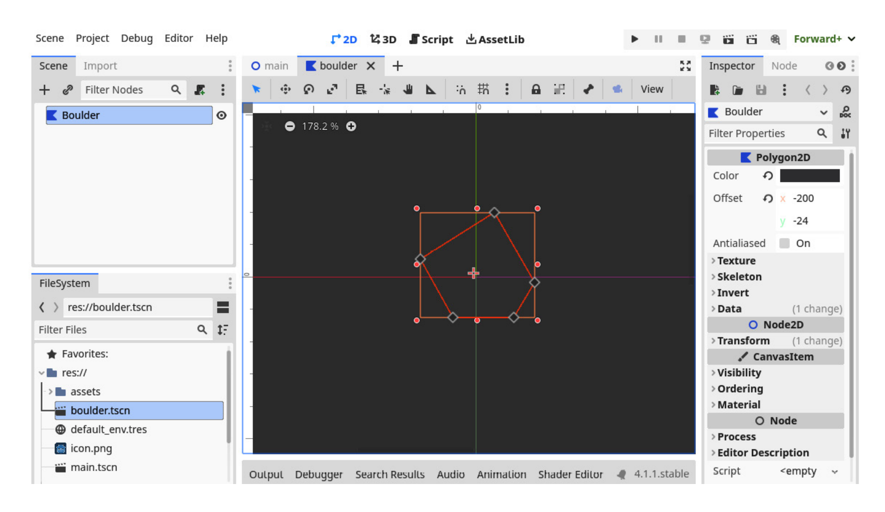


Рисунок 8.8 – Сцена boulder.tscn

Теперь давайте повторно используем эту новую сцену на нашей арене в качестве камня по умолчанию:

1. Вернитесь в сцену main. Уберите все камни со сцены, они нам больше не нужны.
2. Выберите узел **Arena**. Теперь всё, что мы добавим, будет добавлено как дочерний элемент этого узла.
3. Перетащите сцену валуна (boulder.tscn) из дока **Файловая система (FileSystem)** в 2D-редактор. Вы увидите, как всплывает визуальный образ валуна, пока вы его перетаскиваете:

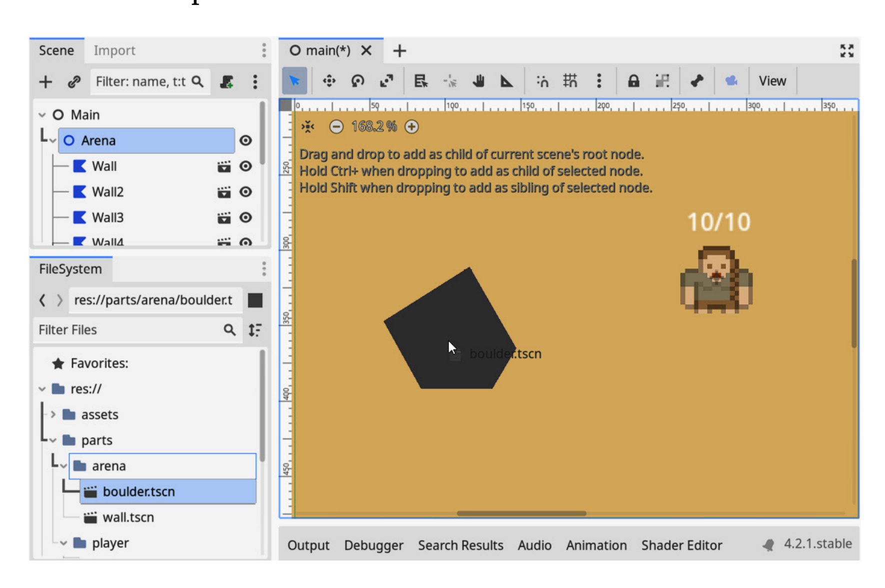


Рисунок 8.9 – Перетаскивание сцены boulder.tscn в дерево сцены Main

Теперь вы можете сделать то же самое для внутренних стен в сцене и заново заполнить арену камнями и стенами, чтобы она не выглядела так пустынно. Однако не делайте то же самое для внешних стен — **OuterWalls** — мы не собираемся использовать их повторно, так что они могут остаться прежними.

При размещении сцен с валунами и стенами вы можете использовать параметры преобразования — Transform, такие как поворот (Rotation), масштаб (Scale) и наклон (Skew), чтобы придать экземплярам разнообразие и не допустить, чтобы они

все выглядели слишком однообразно.

Теперь самое классное — мы можем использовать любую сцену внутри любой другой сцены!

Наличие множества небольших файлов сцен имеет много преимуществ, например, удобство поддержки кода и простота повторного использования (это всего лишь два из них). Но это также усложнит файловую структуру проекта. Из-за этого нам придется подумать о том, как мы организуем все файлы в проекте. Мы сделаем это в следующем разделе.

Организация файлов сцен в проекте

Теперь, когда у нас в проекте стало больше файлов, о которых нужно беспокоиться, нам придется начать проявлять смекалку в отношении того, как мы их организуем. Давайте разделим сцены по разным папкам, которые имеют смысл для нашего проекта. Таким образом, мы всегда будем знать, где что-то найти или сохранить новую сцену.

Добавьте следующие папки в корневую папку нашего проекта:

- **parts**:
    - **environment**
    - **player**
- **screens**:
    - **game**

Папка **parts** будет содержать все сцены, являющиеся частью другой сцены, такие как игрок, стены, враги, предметы коллекционирования, кнопки пользовательского интерфейса и т.д.

screens будет содержать все сцены, которые могут

существовать сами по себе, например, игровой экран, полноэкранные меню, такие как главное меню или меню паузы и т.д. Эти сцены состоят из сцен, которые хранятся в папке **parts**.

В начале проекта я дал вам папку **assets**. Эта папка используется для хранения всех художественных ресурсов, от спрайтов до анимаций и звуков..

Теперь переместите все сцены и скрипты в соответствующие папки, например так:

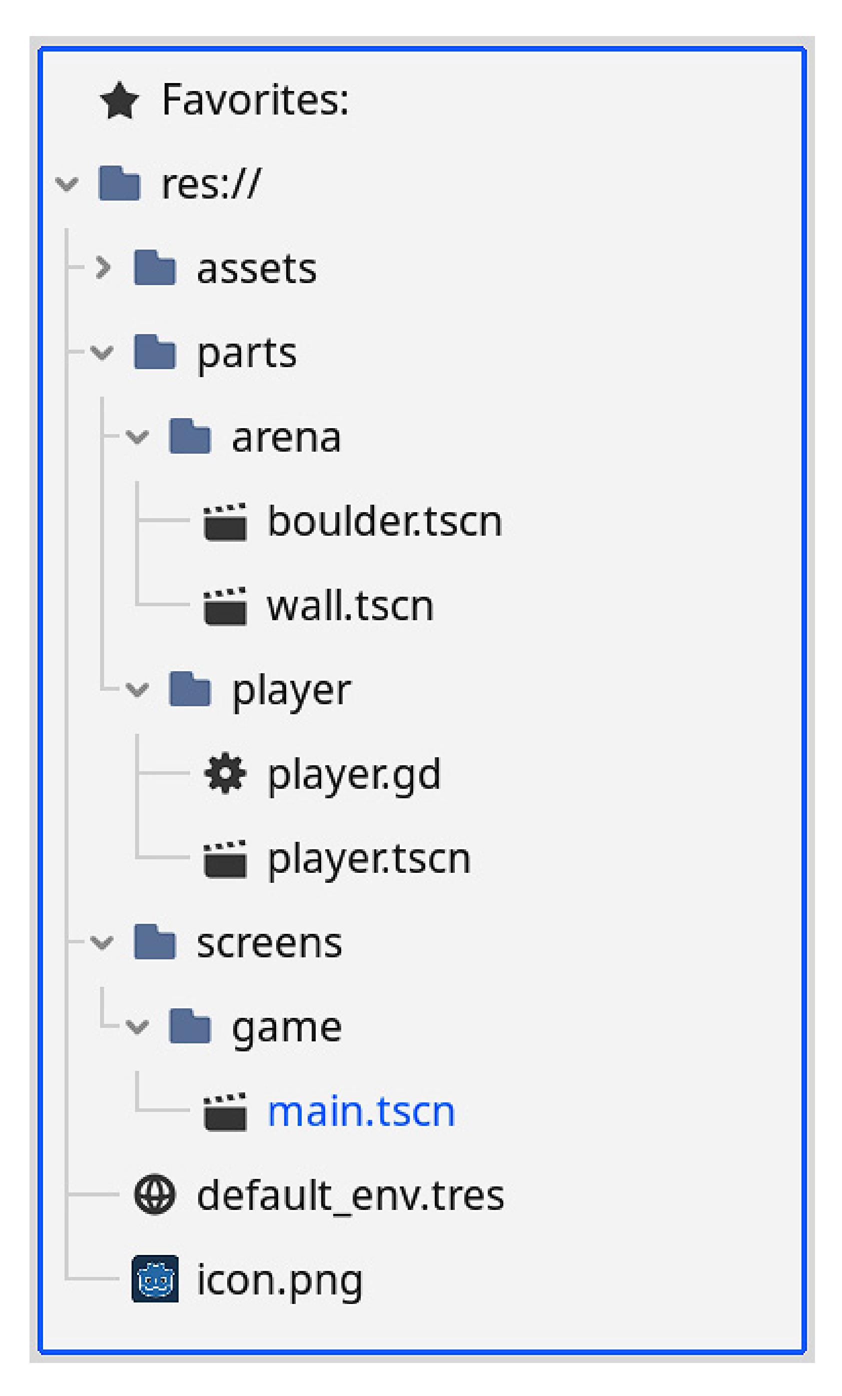


Рисунок 8.10 – Наш проект с улучшенным управлением

файлами

Когда вы посмотрите на другие игровые проекты, созданные в Godot или других игровых движках, вы увидите, что у каждого свой способ организации различных файлов в своём проекте. Мне нравится хранить сцены и скрипты вместе в одной папке, например, потому что большую часть времени вы будете использовать и редактировать их совместно. Однако я буду хранить такие ресурсы, как картинки и звуки, отдельно, потому что их проще повторно использовать в разных сценах.

Со временем вы, вероятно, разработаете собственную организационную структуру для проектов, и это нормально. Что бы вам ни казалось наиболее разумным, это то, что вам следует использовать, пока вы последовательны.

Дополнительные упражнения – заточка топора

1. Используя то, что мы узнали о разделении сцен, попробуйте создать вторую сцену валуна с формой, отличной от первой. Назовите первую сцену валуна **boulder01.tscn**, а вторую — **boulder02.tscn**.

Итоги

Повторное использование частей вашей работы почти всегда является хорошей идеей. В этой главе мы узнали, как повторно использовать целые ветви дерева сцены в качестве отдельных сцен. Это пригодится в следующих главах, поскольку теперь мы можем работать над движением камеры игрока отдельно и создавать столкновения для всех камней или стен одновременно. Но это тема для следующей главы.

Опрос

- Как сохранить ветвь дерева сцены как отдельную сцену??
- Важно ли организовывать наши сцены, скрипты и ресурсы в доке **Файловая система (FileSystem)**? Почему?

9

Камеры, столкновения и подбираемые предметы

Во время игры игрок не хочет думать о камере и её размещении. Камера всегда должна следовать за персонажем игрока и предугадывать, чего игрок хочет добиться, чтобы не загораживать ему обзор.

В более масштабных играх задача команды — создать максимально плавную камеру. В этой главе мы попытаемся сделать то же самое, используя некоторые узлы Godot Engine.

После этого мы запретим игроку проходить сквозь стены и рассмотрим возможность разбрасывания по арене коллекционных предметов, таких как здоровье и деньги.

В этой главе мы рассмотрим следующие основные темы:

- Создание камеры, которая следует за игроком
- Столкновения с валунами и стенами
- Маски столкновения
- Создание унаследованных сцен
- Подключение к сигналам

Технические требования

Как и для каждой главы, окончательный код можно найти в репозитории GitHub в подпапке для этой главы по адресу https://github.com/PacktPublishing/Learning-GDScript-by-Developing-a-Game-with-Godot-4/tree/main/chapter9.

Создание камеры, которая

следует за игроком

На данный момент наш персонаж может бегать, но в какой-то момент он убежит за пределы экрана и потеряется навсегда. Наша внутриигровая камера должна следовать за ним, чтобы игрок знал, где он находится.

К счастью, Godot Engine имеет довольно хорошую систему камер, которую мы можем использовать. Она содержит только основные функции, но это всё, что нам нужно — с некоторыми дополнительными узлами мы сможем добиться очень плавного движения камеры.

Настройка базовой камеры

Для 2D-игр Godot предоставляет узел **Camera2D**.

Откройте сцену **player.tscn** и добавьте узел **Camera2D** и добавьте узел **Player**. Это всё, что нам нужно, чтобы создать базовую камеру, которая следует за игроком::

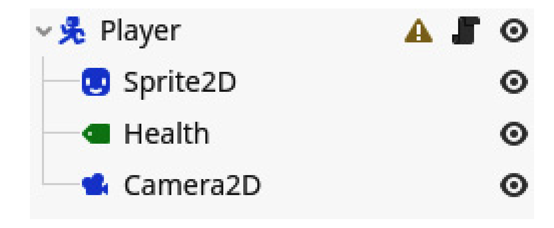


Рисунок 9.1 – Сцена player с добавленным узлом Camera2D

Но эта базовая камера кажется немного жесткой. Она начинает двигаться и останавливаться в тот же момент, когда персонаж делает это. Это кажется не очень естественным. Давайте посмотрим, как это решить.

Добавление полей перетаскивания

Чтобы движение камеры выглядело естественно, мы будем использовать **поля перетаскивания (drag margins)**. Найдите и

включите **Горизонтальное (Horizontal)** и **Вертикальное (Vertical)** перетаскивание в инспекторе камеры в разделе **Drag**:

Drag	
Horizontal Enabled	On
Vertical Enabled	On
Horizontal Offset	0
Vertical Offset	0
Left Margin	0.1
Top Margin	0.1
Right Margin	0.1
Bottom Margin	0.1

Рисунок 9.2 – Настройки поля перетаскивания для узла Camera2D в инспекторе

Теперь камера перемещается только тогда, когда игрок покидает определенную область в середине экрана. Это граница (margin), в пределах которой ничего не происходит. Если вы включите **Отображать границы перетаскивания (Draw Drag Margin)** в инспекторе камеры, вы можете увидеть изображение границ перетаскивания:

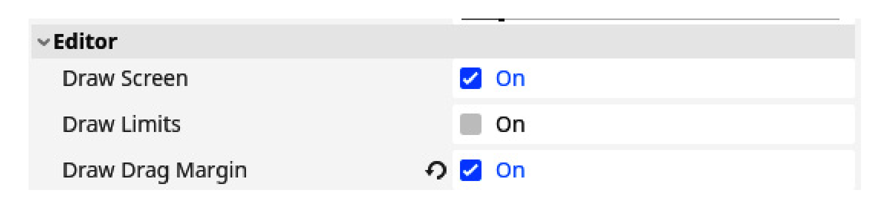


Рисунок 9.3 – Отображение полей перетаскивания в редакторе путем их включения в инспекторе Camera2D

При включённой функции **Draw Drag Margin** вы должны увидеть синие прямоугольники, указывающие, когда камера начнёт двигаться:

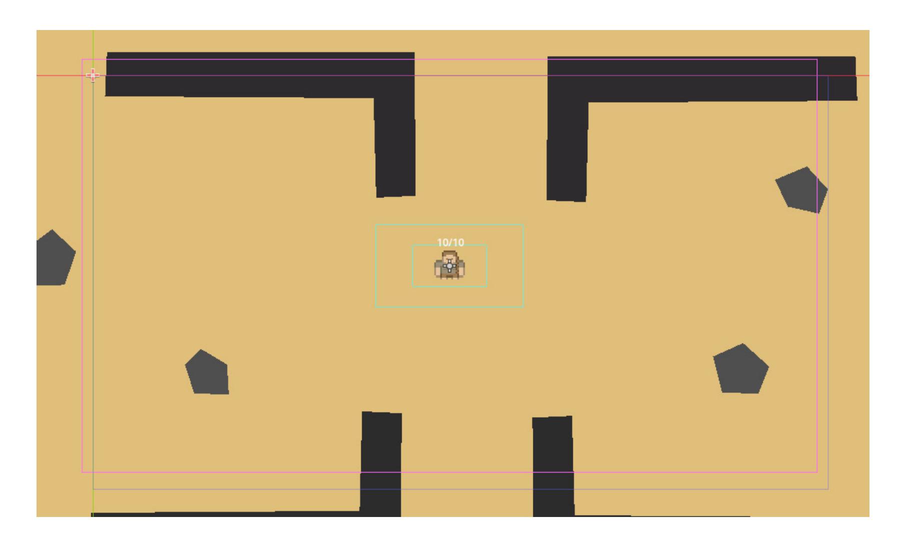


Рисунок 9.4 – Редактор теперь отображает поля перетаскивания светло-голубым цветом

Вы можете немного поиграть с левым, верхним, правым и нижним полями в инспекторе. Я решил установить их все на **0.1**, как показано на *Рисунке 9.2* выше.

Отлично, границы перетаскивания уже выглядят неплохо. Но камера всё ещё начинает движение и останавливается очень резко и, кажется, отстаёт от игрока. Давайте исправим это в следующий раз.

Заставим камеру смотреть вперёд

Движение камеры теперь кажется довольно хорошим. Но что-то не так; нам нужно что-то более фундаментальное, чем приятное, плавное движение. Когда игрок движется, камера тянется позади, показывая, где игрок был, а не куда он идёт.

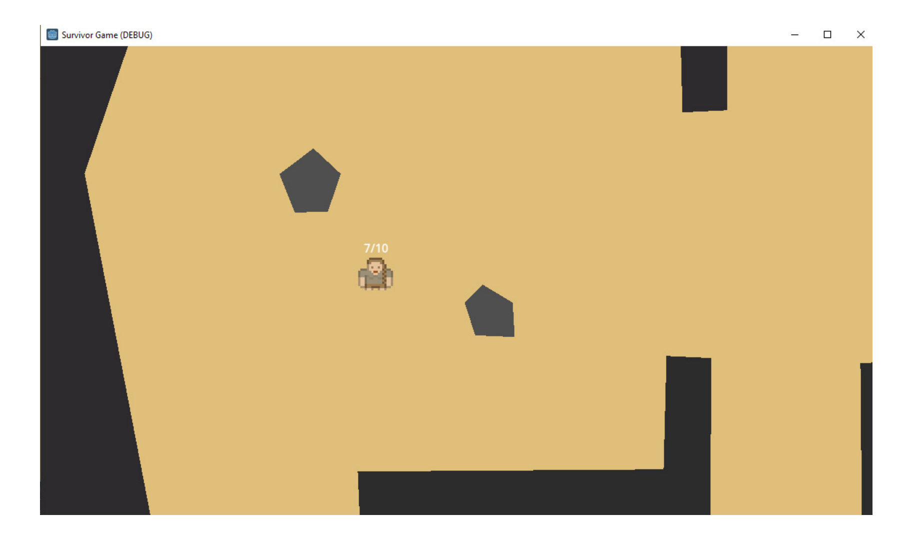


Рисунок 9.5 – Камера отстаёт от игрока и не показывает, куда он идёт

Это не очень здорово; представьте, что вы бежите куда-то и можете смотреть только назад. На самом деле мы хотим, чтобы камера смотрела вперёд в направлении движения игрока. Мы можем сделать это, вместо отслеживания самого игрока, отслеживая точку перед ним. По сути, это как если бы персонаж игрока держал палку для селфи. Итак, выполните следующие действия:

1. Добавьте новый **Node2D** к **Player** и назовите его **CameraPosition**. Это станет точкой, которую мы будем отслеживать вместо самого игрока.
2. Теперь перетащите камеру, которую мы уже создали, на этот **CameraPosition** так, чтобы она стала дочерним элементом нового узла:

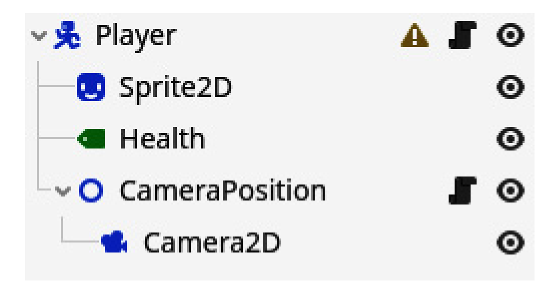


Рисунок 9.6 – Поместите узел Camera2D в отдельный узел с именем CameraPosition

1. Убедитесь, что параметр Position у **Camera2D** и **CameraPosition** установлен на **(0, 0)** в инспекторе:



Рисунок 9.7 – Убедитесь, что узлы Camera2D и CameraPosition находятся в положении (0, 0)

1. Добавьте скрипт в **CameraPosition** и назовите его **camera_position.gd**.

Этот скрипт будет удерживать позицию перед камерой. Мы можем сделать это на основе скорости — **velocity**, которую имеет персонаж игрока.

Полностью скрипт выглядит так:

```
extends Node2D
@export var camera_distance: float = 200
@onready var _player: CharacterBody2D = get_parent()
func _process(_delta):
   var move_direction: Vector2 = _player.velocity.normal
   position = move_direction * camera_distance
```

Сначала мы определяем экспортную переменную, с которой мы можем работать, под названием **camera_distance**. Это будет расстояние перед игроком, на котором мы будем удерживать камеру во время движения игрока.

С помощью **_physics_process()**, который выполняется для каждого физического кадра игры, мы вычисляем положение точки камеры. Помните, что это положение относительно узла **Player**, поэтому положение **(0, 0)** находится прямо там, где находится игрок. Идея состоит в том, чтобы взять направление, в котором движется игрок, и умножить его на **camera_distance**.

Направление, в котором движется игрок, можно вывести из скорости — **velocity** игрока. Итак, сначала мы получаем узел player с помощью функции **get_parent** и кэшируем его в переменной **_player**. Эта функция возвращает родительский узел узла, в данном случае узел **Player**, поскольку **CameraPosition** является прямым дочерним элементом этого узла.

Затем, чтобы получить направление, в котором движется игрок, мы нормализуем этот вектор **velocity**. Нормализация вектора, как мы видели в *Главе 7*, означает, что вы берёте весь вектор и делаете его длиной **1**. Таким образом, весь вектор имеет длину **1** пиксель. Это оставит нам направление скорости **velocity** без его длины. Теперь мы можем легко масштабировать это направление до любой нужной нам длины, умножив его на **camera_distance** , чтобы определить положение — **Position** узла **CameraPosition**. Если вы запустите игру сейчас, вы увидите, что **CameraPosition** делает то , что вы хотите, и заставляет камеру смотреть вперед в направлении движения персонажа. Но оно всё ещё немного дёргается, поэтому давайте наконец сгладим это.

Сглаживание взгляда вперёд

Проблема в том, что камера начинает двигаться очень внезапно и резко останавливается. Это происходит потому, что **CameraPosition** , который мы создали, прыгает довольно быстро. Чтобы исправить эту проблему, мы должны сделать движение самого **CameraPosition** плавным.

Сначала добавьте новую экспортируемую переменную, например:

```
@export var position_interpolation_speed: float = 1.0
```

Теперь давайте изменим функцию **_process()** скрипта **camera_position.gd** следующим образом:

```
func _physics_process(delta):
    var move_direction: Vector2 = _player.velocity.normal
    var target_position: Vector2 = move_direction * camer
    position =  position.lerp(target_position, position_i
```

Мы делаем в основном то же самое, что и в предыдущем разделе, но на этот раз мы сохраняем позицию, к которой мы хотим привести **CameraPosition**, в переменной **target_position**. Затем, в следующей строке, мы вычисляем **фактическую позицию**.

Для вычисления позиции мы используем новую функцию — **lerp()**. Это сокращение от **линейной интерполяции (linear interpolation)**. Эта функция работает почти так же, как **move_towards()**, которую мы использовали в *Главе 7*. Однако, в то время как **move_towards()** перемещает позицию на определенное количество пикселей в направлении целевой позиции, **lerp** перемещает позицию в направлении **target_position** в соответствии с процентом между ними двумя. Этот процент является последним аргументом в функции и выражается значением от **0.0** до **1.0**, где **0.0** это **0%**, а **1.0** это **100%**.

Итак, предположим, что мы хотим переместиться на **50%** между позицией и её целью, тогда результирующая позиция будет находиться точно посередине между двумя точками.

Этот процесс называется **линейной интерполяцией**, потому что мы интерполируем между двумя значениями линейно. Способ, которым мы используем линейную интерполяцию в нашем скрипте положения камеры, заключается в том, чтобы немного двигаться к целевой позиции каждый кадр.

Процент линейной интерполяции, который я выбрал, был **5.0 * delta**. Я поместил это значение в экспортированную переменную, чтобы мы могли легко настроить его из редактора. Поскольку delta — это время между двумя кадрами, результат этого произведения очень мал и должен привести к интерполяции около **10%** на кадр. Мы умножаем на **delta**, потому что, как и для скорости движения игрока, мы хотим, чтобы скорость камеры не менялась на более быстрых или

медленных компьютерах, работающих с более высокой или низкой частотой кадров. Мы также говорили о независимых от частоты кадров вычислениях в *Главе 7*, когда заставляли персонажа двигаться.

Вы можете поиграть со скоростью интерполяции, изменив **position_interpolation_speed** на любое другое значение через инспектор.

В этом разделе мы много узнали о создании полезной и плавной камеры, которая показывает, куда движется игрок. Как упоминалось во введении к этой главе, в высокобюджетных играх есть целые команды, которые работают только над камерой. Но, используя некоторые хитрые трюки, мы добились довольно хорошей работы камеры в нашей маленькой игре. Теперь мы изменим скорость и убедимся, что игрок не бежит сквозь стены, добавив обнаружение столкновений.

Столкновения

Теперь, когда у нас есть камера, давайте обратим внимание на столкновения. На данный момент у нас есть визуальные представления для хорошей арены, включая стены и камни, но они не ведут себя как настоящие. Персонаж игрока может просто пробежать сквозь них, как будто они сделаны из воздуха, а не из твёрдого вещества.

Как и в случае с движением игрока, мы можем решить эту проблему, используя встроенный физический движок. Давайте начнем с рассмотрения различных физических тел, имеющихся в нашем распоряжении.

Различные физические тела

Для персонажа игрока мы использовали физическое тело **CharacterBody2D**. Но это не единственный тип, который поставляется с физическим движком Godot. Есть ещё несколько:

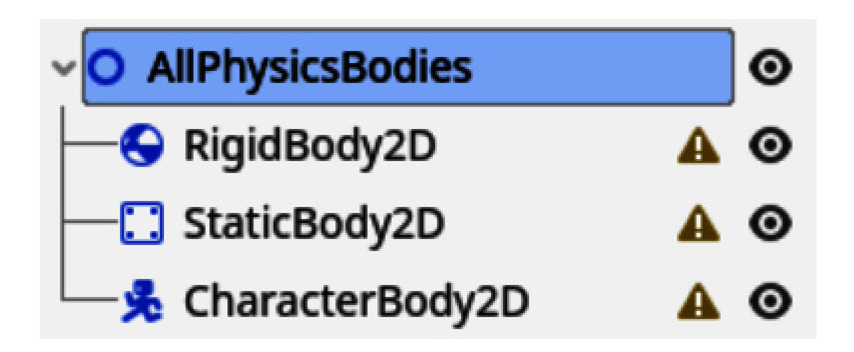


Рисунок 9.8 – Три различных физических тела, отображаемые в дереве сцены

Для 2D-игр доступны три различных типа физических тел. Каждое из них имеет своё применение в игре или физическом моделировании. Давайте рассмотрим каждое из них.

RigidBody2D

Твёрдое тело (rigid body) — это физический объект, который полностью смоделирован. Твёрдые тела полностью зависят от внешних физических сил и столкновений. Вы не должны управлять ими напрямую.

Важное примечание

Эти тела называются твёрдыми, потому что само тело не деформируется.Их используют для моделирования твёрдых объектов от автомобилей до костей и стен. **StaticBody2D** и **CharacterBody2D** также не могут деформироваться и поэтому также являются твёрдыми телами, но больше в математическом смысле этого слова, а не в том, как они реализованы в движке.

Мы не можем напрямую управлять твердым телом; они полностью управляются физическим движком, который решает, как оно движется и как применяются скорости и силы. Единственный способ управлять твердым телом — это применять к нему внешние силы. Это как ударить по мячу для гольфа клюшкой. При достаточной практике и тонкой настройке вы можете направить мяч в направлении лунки, но поднять его и бросить туда, хотя это и проще, не вариант.

Моделирование нежёстких тел, также известных как мягкие тела, обычно сложнее осуществить математически и производительно в игре. Мягкие тела могут быть губками, резиновыми объектами, которые деформируются, желе и т.д. Существуют способы их моделирования в рамках моделирования твёрдого тела. но делать этого не рекомендуется. В 3D есть узел **Softbody3D**, но для 2D он не подходит.

StaticBody2D

Статическое тело (static body)— это физическое тело, которое остаётся статичным, то есть оно не движется и не может быть подтолкнуто внешними силами. Это самое простое физическое тело, с которым можно иметь дело, и оно идеально подойдёт для создания наших стен и камней.

CharacterBody2D

Тело персонажа (character body) — это физическое тело, которым мы можем управлять через код. **RigidBody2D**, как мы видели ранее, полностью управляется физическим движком. Это затрудняет управление, чтобы заставить его делать то, что вы хотите.

Тело персонажа, с другой стороны, дает нам хорошую золотую середину. Как и в скрипте **player.gd**, мы должны сами рассчитать скорость — **velocity** и вызвать **move_and_slide()**. Но физический движок всё ещё помогает нам с столкновениями и вычислением того, куда тело должно двигаться на основе скорости.

Дополнительная информация

В документации Godot также имеется отличное описание различных физических тел и того, как их можно использовать: https://docs.godotengine.org/en/stable/tutorials/physics/physics_introduction.html#collision-objects.

Это были три типа физических тел, доступных в Godot. Но на

самом деле есть четвертый физический объект, который не является телом. Давайте рассмотрим узел **Area2D**.

Узел Area2D

Три физических тела, которые мы только что видели, сталкиваются друг с другом и реагируют на это столкновение или заставляют другие тела реагировать на него. По сути, их движения обрабатываются физическим движком.

Последний физический объект, **Area2D**, только обнаруживает и влияет на другие физические объекты. Он не подвергается физическим расчетам, как для движения. Но он может обнаружить, перекрывает ли его другой физический объект, и выдаёт сигнал, когда эти другие физические объекты входят или выходят.

Мы воспользуемся этой функцией ближе к концу главы, чтобы игрок мог подбирать зелья здоровья, когда подходит к ним.



Рисунок 9.9 – Четыре различных физических объекта, отображаемые в дереве сцены

Теперь, когда вы знаете о трёх типах физических тел, доступных в настоящее время, мы можем начать использовать их для создания правильных столкновений.

Добавление формы столкновения к

узлу игрока

С тех пор, как мы создали узел игрока в *Главе 6*, рядом с ним был этот маленький оранжевый треугольник. Когда мы наводим на него курсор, подсказка объясняет нам, что у этого узла отсутствует форма. Это имеет смысл, поскольку физический движок не может обнаружить столкновения, если он не знает, какой формы физическое тело.

Давайте разберёмся с этим маленьким предупреждением:

1. Найдите и добавьте узел **CollisionShape2D** в качестве дочернего узла узла **Player**:

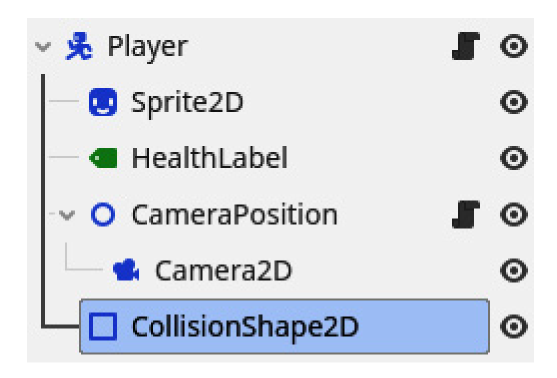


Рисунок 9.10 – Добавление CollisionShape2D к сцене игрока

1. Выберите этот недавно созданный узел **CollisionShape2D** и щёлкните пустое поле **Форма (Shape)**, чтобы открыть ниспадающее меню с различными формами.
2. Выберите опцию **CapsuleShape2D**.

CollisionShape2D	
Shape	CapsuleShape2D
Radius	10 px
Height	30 px
Custom Solver Bias	0
› **Resource**	(1 change)

Рисунок 9.11 – Выберите CapsuleShape2D в качестве формы CollisionShape2D

1. На экране появится капсулоподобная синяя фигура. Это форма физического тела. Используйте оранжевые круги на периферии фигуры, чтобы изменить её размер и постарайтесь покрыть большую часть спрайта игрока:



Рисунок 9.12 – Убедитесь, что CapsuleShape2D закрывает спрайт игрока

Узел **CollisionShape2D** сам по себе не имеет формы, но он будет содержать ее для нас. Вот почему нам пришлось добавить

её к свойству **Shape**.

Другие формы, которые интересны как формы столкновений, это **RectangleShape2D** и **CircleShape2D**. Другие используются в особых ситуациях, например, для очень тонких или разрозненных объектов, так что пока не стоит слишком беспокоиться о них.

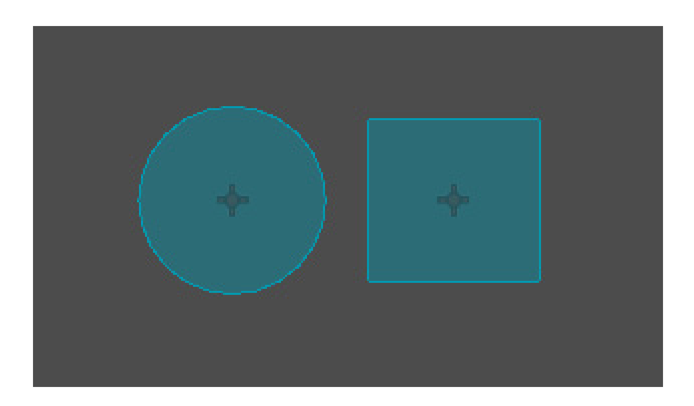


Рисунок 9.13 – CircleShape2D и RectangleShape2D

Запуск игры сейчас не приведёт к столкновению игрока с валунами или стенами, просто потому, что сначала нам также нужно будет добавить физические тела и формы в сцены ещё и для них.

Создание статических тел для валунов

В разделе *Различные физические тела* мы узнали, что узлы **StaticBody2D** не двигаются; это звучит идеально для валуна. Итак, давайте сделаем их твёрдыми:

1. Зайдите в нашу сцену **boulder.tscn**.
2. Добавьте узел **StaticBody2D** под корневым узлом:

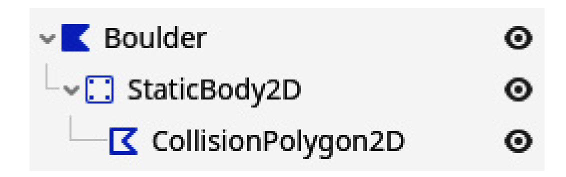


Рисунок 9.14 – Добавление StaticBody2D с CollisionPolygon2D в качестве дочернего элемента сцены валуна

1. Добавьте узел **CollisionPolygon2D** под это недавно созданное статическое тело.
2. Теперь добавьте точки в полигон столкновения, щёлкнув в 2D-редакторе. Попробуйте полностью покрыть валун:

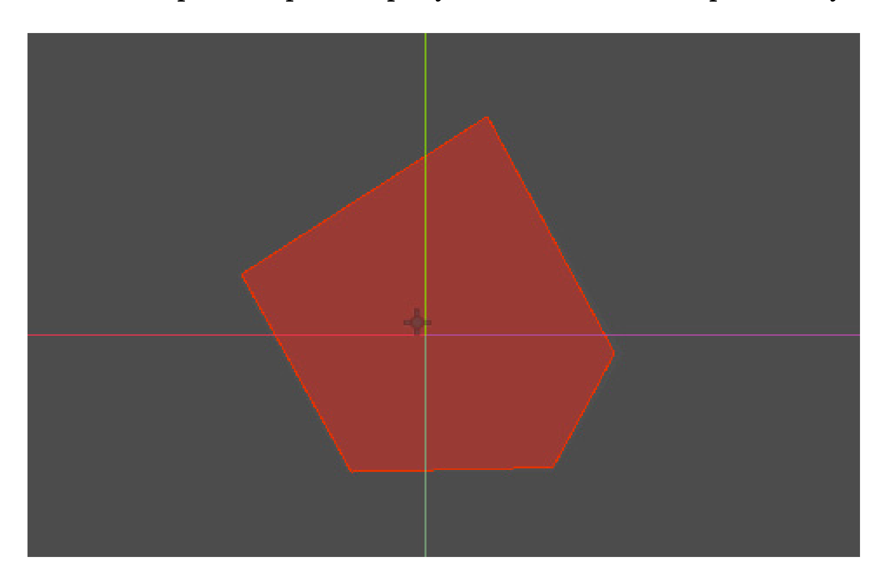


Рисунок 9.15 – Покрытие валуна с помощью PolygonShape2D

Обратите внимание, что мы используем другой вид формы столкновения. Теперь мы используем **CollisionPolygon2D**. Эта форма позволяет нам определять нашу собственную произвольную форму. Преимущество в том, что мы можем создать любую форму, которая нам нравится. Недостаток в том, что произвольные полигоны немного медленнее обрабатываются физическим движком. Но это не должно быть большой проблемой в нашей игре, потому что у нас не будет тысяч объектов, требующих сложных физических вычислений.

Теперь, когда мы знаем, как создавать статические тела, мы можем сделать то же самое для других статических объектов в нашей игре, например, для стен.

Создание статических тел для стен

Давайте сделаем нечто подобное, добавив столкновение со стенами в игре:

1. Откройте **wall.tscn**.
2. Добавьте узел **StaticBody2D**.



Рисунок 9.16 – StaticBody2D стены имеет два дочерних элемента CollisioniShape2D

1. Вместо использования **CollisionPolygon2D** добавьте два узла **CollisionShape2D**.
2. Присвойте каждому из них **RectangleShape2D** в свойстве **Shape**.
3. Теперь убедитесь, что комбинация этих двух фигур покрывает стену:

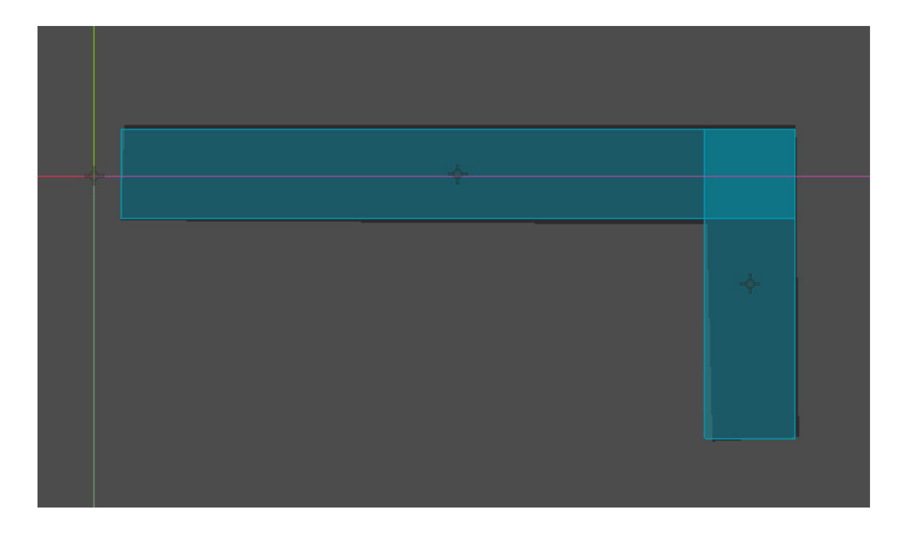


Рисунок 9.17 – Покрытие стены с помощью двух RectangleShape2D

Из этого примера вы можете видеть, что одно физическое тело может фактически содержать несколько форм столкновений. Это очень полезно при построении сложных форм без необходимости прибегать к **CollisionPolygon2D**. Хотя мы использовали две прямоугольные формы, мы могли бы использовать две разные формы, если бы захотели, даже комбинируя обычные формы и многоугольники. Мы можем добавить столько форм под одно физическое тело, сколько пожелаем.

В этом разделе мы узнали, как использовать различные физические тела для обнаружения столкновений и убедиться, что игрок не проходит сквозь стены и валуны. В следующем разделе мы расширим эти знания, чтобы также использовать физический движок для определения того, находится ли игрок в определённой области или нет.

Создание предметов коллекционирования

Теперь давайте создадим несколько коллекционных предметов, которые наш герой сможет подобрать. Мы создадим два разных коллекционных предмета:

- Зелье здоровья, которое восполнит здоровье персонажа.
- Монета, которая добавит единицу золота к деньгам игрока.

Начнём с создания базового коллекционного предмета, на основе которого мы сможем легко реализовать два разных поведения, которые будут иметь два коллекционных предмета.

Создание базовой сцены коллекционного предмета

Базовая сцена и класс, которые мы построим для наследования каждого конкретного коллекционного предмета, очень важны. Они должны охватывать вариант использования всех других коллекционных предметов, которые мы хотим создать. Итак, начнем:

1. Создайте новую сцену с именем **collectible.tscn** в новой папке **parts/collectibles**.
2. Настройте сцену, как показано на *Рисунке 9.18*:

    1. Сделайте корневой узел **Node2D** и назовите его **Collectible**.
    2. Добавьте узлы **Area2D** и **Sprite2D** в качестве прямых потомков.
    3. Добавьте **CollisionShape2D** к узлу **Area2D**.

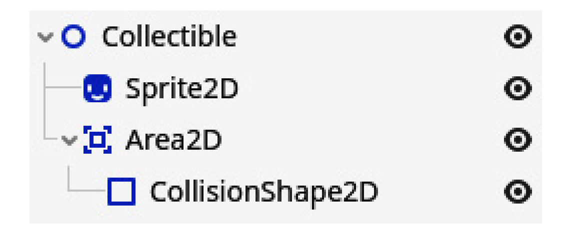


Рисунок 9.18 – Базовая структура сцены для наших предметов коллекционирования

1. Заполните свойство **Shape** объекта **CollisionShape2D** значением **CircleShape2D**.
2. Обновите радиус круга до **25** пикселей.

Мы используем новый тип узла — **Area2D**. Узел **Area2D** может обнаруживать столкновения, когда другое физическое тело или область входит в его форму. Поскольку мы используем этот физический объект, узел **Area2D** не будет выполнять какую-либо физику и не будет влиять на физику другого физического тела. Узлы **Area2D** используются для обнаружения того, перекрывают ли другие тела или области их форму столкновения. Мы будем использовать эту функциональность для обнаружения того, перекрывает ли персонаж игрока предмет коллекционирования, потому что когда это происходит, нам приходится выполнять код, связанный с

предметом коллекционирования.

Имея готовую базовую коллекционную сцену, мы можем легко наследовать от неё в следующем разделе.

Наследование от базовой сцены

Если вам интересно, почему мы до сих пор не добавили текстуру к сцене коллекционных предметов, то это потому, что мы хотим сделать это для определённых предметов коллекционирования, таких как зелье здоровья и монета, а не для базы.

Давайте создадим конкретный предмет коллекционирования:

1. Щёлкните правой кнопкой мыши по сцене **collectible.tscn** в файловом менеджере и выберите **Новая вложенная сцена (New Inherited Scene)**. Эта вложенная сцена наследует узлы исходной сцены.

FileSystem
res://parts/collectib
Filter Files
Favorites:
res://
assets
parts
camera
collectible
collectible.tscn
environment
player
screens
default_env.tres
icon.png

Open Scene
New Inherited Scene
Set As Main Scene
Instantiate
Edit Dependencies...
View Owners...
Create New
Copy Path Ctrl+Shift+C
Copy UID
Rename... F2
Duplicate... Ctrl+D
Move/Duplicate To...
Delete Delete
Add to Favorites
Show in File Manager
Open in External Program

Рисунок 9.19 — Щелчок правой кнопкой мыши по файлу collectible.tscn и выбор «Новая вложенная сцена»

1. Откроется новая сцена. Переименуйте корневой узел в **HealthPotion**:

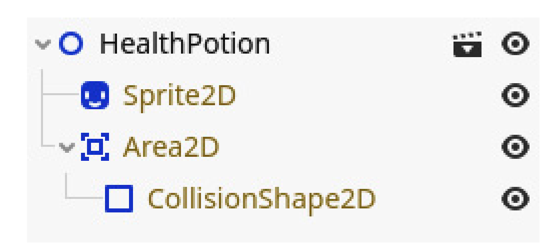


Рисунок 9.20 – Унаследованные узлы выделены серым цветом

3. Сохраните эту вложенную (унаследованную) сцену как **health_potion.tscn** в той же папке, что и **collectible.tscn**, то есть в **parts/collectibles**.
4. Теперь добавьте **HealthPotion.png** из **assets/visual/collectibles** в качестве текстуры к узлу **Sprite2D**.
5. Спрайт немного маловат, поэтому установите масштаб (Scale) на **(2, 2)**, как мы это делали для спрайта игрока в *Главе 6*:

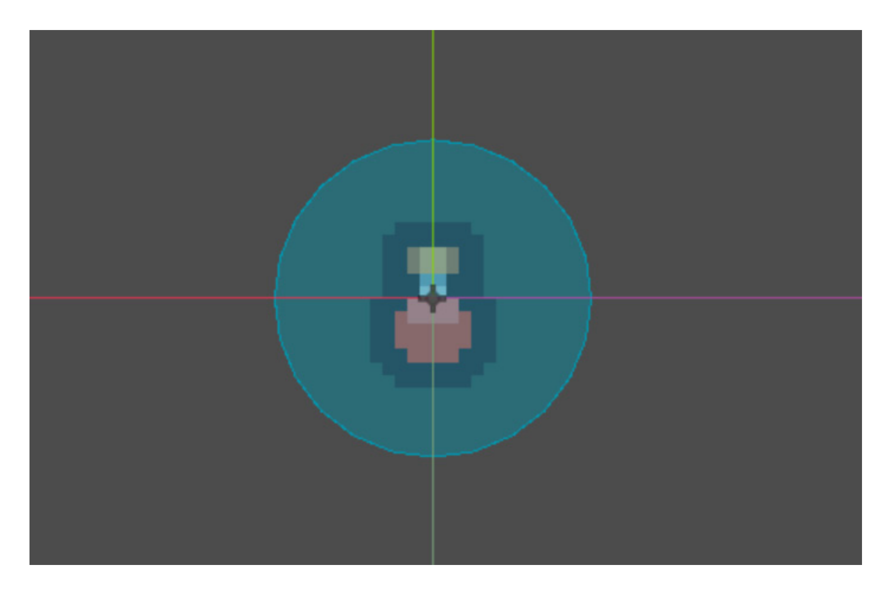


Рисунок 9.21 – Вот как наше коллекционное зелье здоровья должно выглядеть в редакторе

Вы можете увидеть, что все узлы, за исключением корневого узла, серые, как на *Рисунке 9.22*. Причина в том, что эти узлы управляются сценой, от которой мы наследуемся, в данном случае сценой **collectible.tscn**.

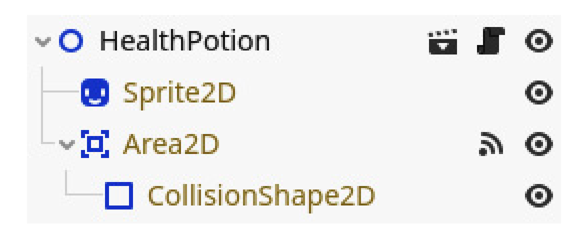


Рисунок 9.22 – При наследовании сцены унаследованные узлы

отображаются серым цветом

Попробуйте сами

В качестве эксперимента вернитесь к сцене **collectible.tscn**, переместите узел спрайта в другое место и сохраните. Если вы теперь посмотрите на сцену **health_potion.tscn**, вы увидите, что и в ней спрайт переместился в то же место.

С помощью техники вложенных сцен (наследования) мы можем легко выстроить функциональность коллекционных предметов без необходимости изменять сцену каждого коллекционного предмета по отдельности или копирования-вставки. Мы можем просто определить базовую структуру и функциональность один раз.

Теперь наша базовая сцена с зельем здоровья готова и мы можем добавить ей логику. Во-первых, нам нужно знать, когда игрок достаточно приблизится, чтобы подобрать коллекционный предмет. Мы узнаем, как это сделать, в следующем разделе.

Подключение к сигналу

В разделе *Создание базовой сцены коллекционного предмета* я говорил, что мы собираемся использовать узел **Area2D** для определения момента, когда в ней появится физическая форма игрока, и мы узнаем, что коллекционный предмет следует подобрать.

Для реализации нам понадобится ещё одна концепция движка Godot: **сигналы (signals)**. Все узлы могут испускать сигналы. Сигналом может быть что-то вроде *“a physics body entered my shape” — "физическое тело вошло в мою форму"*. Мы могли бы прослушивать или подключаться к этому сигналу и запускать фрагмент кода всякий раз, когда сигнал испускается.

Теперь мы сделаем это для сигнала, который выдаёт узел **Area2D**, когда физическое тело входит в форму столкновения:

1. Перейдите на сцену **collectible.tscn**.

2. Добавьте пустой скрипт в корневой узел. Чтобы подключиться к сигналу, нам сначала нужен скрипт. Обязательно удалите весь код внутри скрипта, за исключением первой строки, которая говорит, что скрипт расширяет **Node2D**. Сохраните скрипт как **collectible.gb**.
3. Теперь выберите узел **Area2D**. На правой панели, где мы обычно видим инспектор узла, также есть вкладка **Узел (Node)**. Щёлкните по ней.

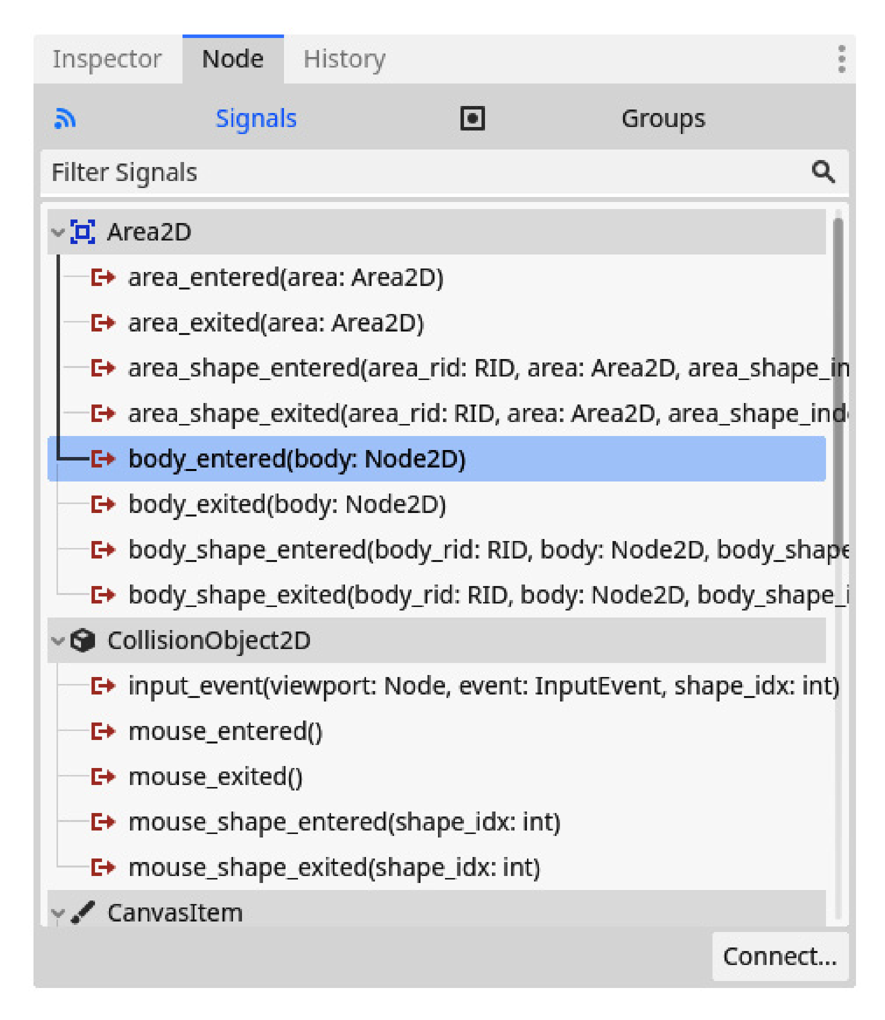


Рисунок 9.23 – Список сигналов, которые может испускать узел Area2D; body_entered – тот, который нам нужен

1. Эта вкладка показывает нам различные сигналы, которые может выдавать узел. Тот, к которому мы хотим подключиться, называется **body_entered**, потому что он испускается с момента, когда физическое тело входит в узел **Area2D**. Выберите этот сигнал и нажмите кнопку **Присоединить… (Connect)** в правом нижнем углу.
2. Появляется модальное окно, спрашивающее нас, к какому узлу в текущей сцене мы хотим подключить этот сигнал. Корневой узел **Collectible** должен быть уже выбран, поэтому просто нажмите кнопку **Присоединить… (Connect)**.

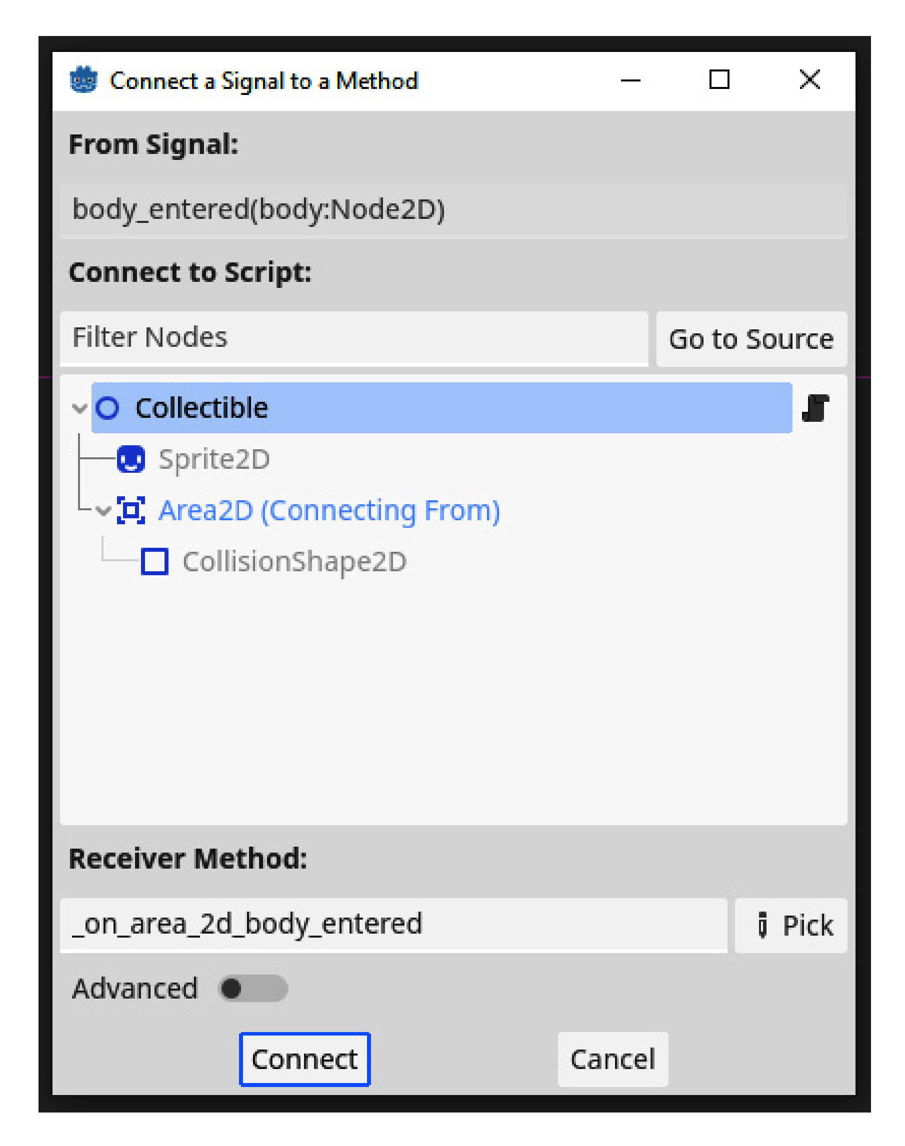


Рисунок 9.24 – Выбор узла, с которым мы хотим соединить сигнал, в нашем случае это узел Collectible

1. Теперь мы перейдём непосредственно к скрипту **collectible.gb**, и вы увидите, что для нас была добавлена новая функция **_on_area_2d_body_entered**:

```
extends Node2D


func _on_area_2d_body_entered(body):
	pass # Replace with function body.

```

Рисунок 9.25 – Новая функция будет автоматически создана для нас после подключения сигнала

Подключение сигнала выполнено. Теперь каждый раз, когда узел **Area2D** выдаёт сигнал **body_entered**, будет выполняться функция **_on_area_2d_body_entered** этого коллекционного предмета.

Также обратите внимание, что сгенерированная функция имеет параметр **body**. Это объект body, который пересекает **Area2D**, например персонаж игрока — player. Сигналы могут давать некоторый контекст, когда они передаются в форме этих параметров. Разные сигналы имеют разные параметры, а большинство вообще не имеют параметров.

Написание кода для предметов коллекционирования

Теперь мы наконец-то напишем настоящий код, который даст игроку несколько новых очков здоровья при подборе зелья здоровья, хотя, честно говоря, их будет не так уж много. Давайте напишем код, необходимый для того, чтобы наше зелье здоровья заработало:

1. Сначала вернемся к скрипту **collectible.gd**. Сделаем этот скрипт именованным классом, добавив строку, определяющую имя класса вверху:

```
class_name Collectible
```

Важное примечание

Создание именованного класса с помощью **class_name** здесь не является на 100% необходимым, но присвоение имен классам, от которых вы собираетесь наследовать, является хорошей практикой.

1. Теперь в сцене **health_potion.tscn** щёлкните правой кнопкой мыши по корневому узлу и выберите **Расширить скрипт (Extend Script)**.
2. Сохраните новый скрипт как **health_potion.gd** в той же папке, что и **health_potion.tscn**. Это создаст скрипт, который наследуется от класса **Collectible** и назначит его узлу **HealthPotion**.

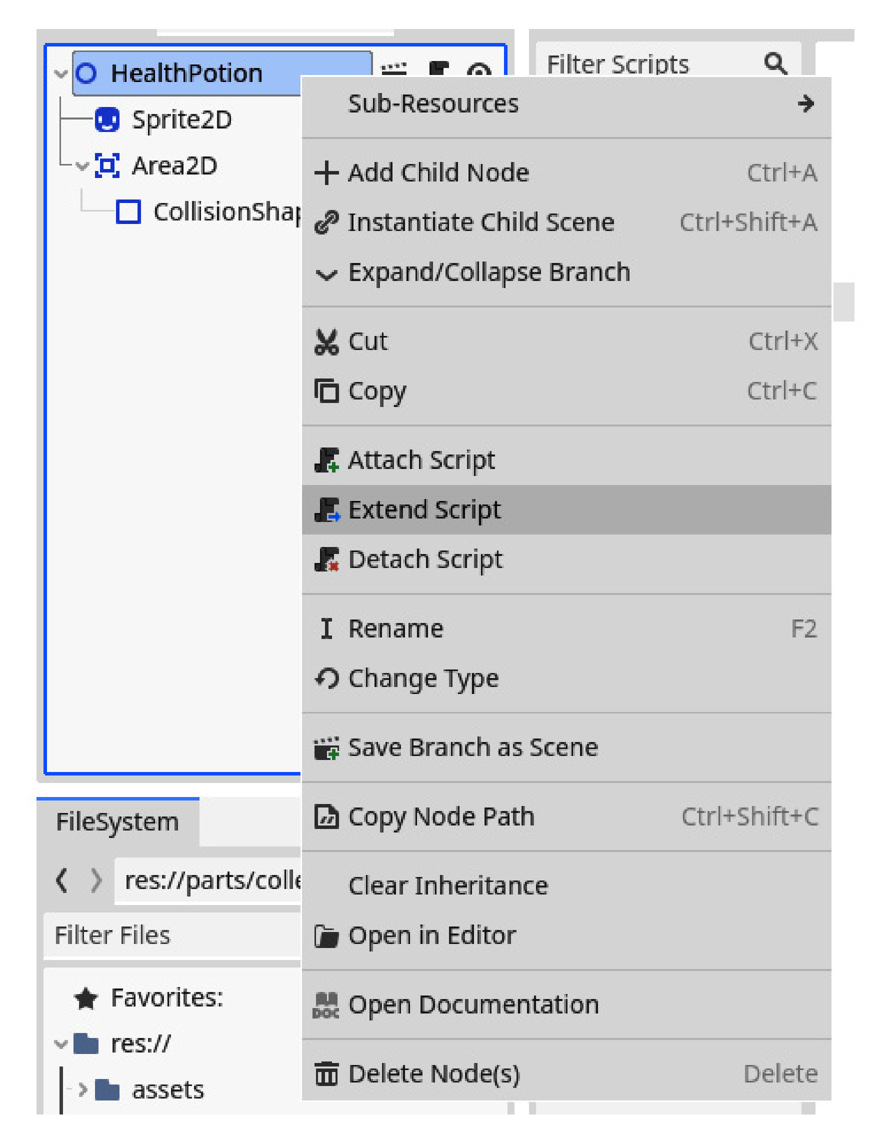


Рисунок 9.26 – Щелчок правой кнопкой мыши по узлу HealthPotion и выбор Расширить скрипт (Extend Script)

1. Затем переопределите функцию **_on_area_2d_body_entered()**, определив новую, например так:

```
func _on_area_2d_body_entered(body):
```

```
        body.health += 5
        queue_free()
```

Эта функция используется в классе **Collectible** для подключения к сигналу **body_entered**. Переопределяя её здесь, мы фактически заменяем функцию, которая будет выполнена.

Вы можете видеть, что мы берем тело (body), предоставленное в качестве аргумента, и просто обновляем его значение здоровья, добавляя **5**.

Последняя строка вводит новую функцию, которую мы можем вызывать на узлах: **queue_free()**. Эта функция поставит узел в очередь на удаление, чтобы движок знал, что его можно удалить из дерева сцены. Движок удалит узел в конце текущего кадра.

Давайте попробуем это! Вернитесь на главную сцену и добавьте зелье здоровья куда-нибудь, перетащив сцену в любое место арены:

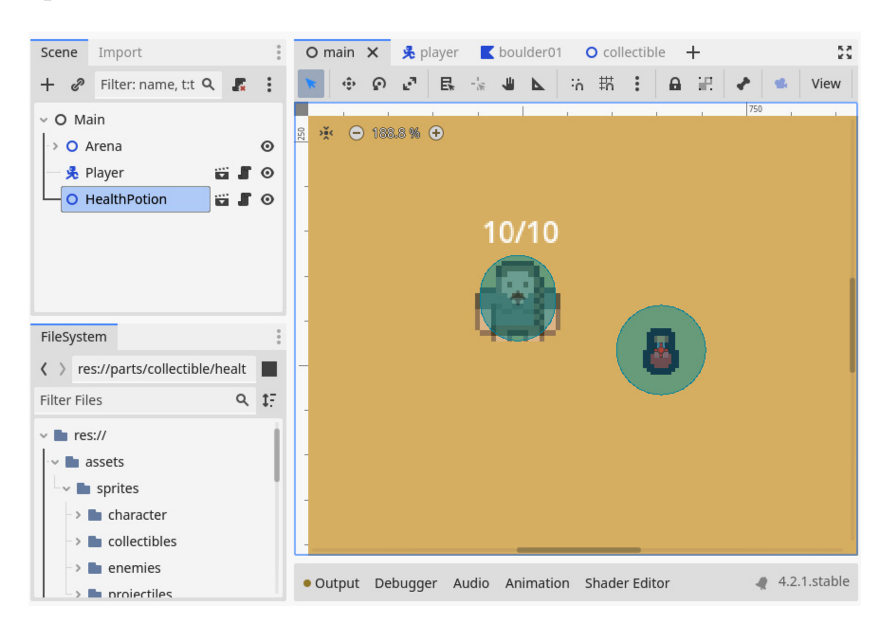


Рисунок 9.27 – Добавление HealthPotion на главную сцену

Если вы положите зелье здоровья куда-нибудь, в место без

валунов или стен, вы сможете подойти туда персонажем игрока и поднять его. Но если вы положите зелье здоровья слишком близко к валуну или стене, вы получите ошибку! О нет! Давайте узнаем, как решить эту проблему дальше.

Использование слоев столкновений и масок

Есть одна проблема! Теперь сигнал будет выдаваться для каждого физического тела, которое входит в **Area2D** коллекционного предмета, даже для валунов и стен. Но мы хотим активировать функционал только тогда, когда наш игрок входит в область.

К счастью, мы можем активировать обнаружение перекрытия только для определённых тел, используя слои столкновений (layers) и маски (masks).

Знакомство со слоями и масками столкновений

Если вы выберете узел **Area2D** в сцене **collectible.tscn**, вы увидите свойства **Слой столкновений (Collision Layer)** и **Маска столкновений (Collision Mask)** в инспекторе. Эти два свойства определяют, какие другие физические тела и области могут взаимодействовать с областью **Area2D**.

- **Слои столкновений (Collision Layers)** определяют, в каком слое находится физический объект, и могут ли они быть обнаружены другими физическими объектами.
- **Маски столкновений (Collision Mask)** определяют, какие слои этот физический объект будет просматривать для обнаружения столкновений.

Рисунок 9.28 – Существует 32 отдельных слоя столкновений и масок

Это означает, что слои столкновений используются для того, чтобы сообщать другим телам и областям о вашем существовании, в то время как маска столкновений используется для обнаружения других тел и областей. Обратите внимание, что они не обязательно должны быть одинаковыми. Слои могут отличаться от маски, и одно тело или область могут быть активны в нескольких слоях и могут просматривать несколько масок.

Каждый слой столкновений имеет номер, связанный с ним, но мы можем дать им имя, которое будет легче читать людям. Мы сделаем это в следующем разделе.

Именование слоёв столкновений

Что мы собираемся сделать, так это использовать один слой, **слой номер 1**, как слой для столкновений со стенами и другой слой, **слой номер 2**, для обнаружения предметов коллекционирования. Поскольку сложно и неинформативно говорить о **слое номер 1** и **слое номер 2**, мы можем дать слоям имена в редакторе Godot. Это поможет нам в дальнейшей работе:

1. Откройте **Настройки проекта (Project Settings)**.

2. Перейдите в раздел **Имена слоя (Layer Names) | 2D Физика (2D Physics)**:

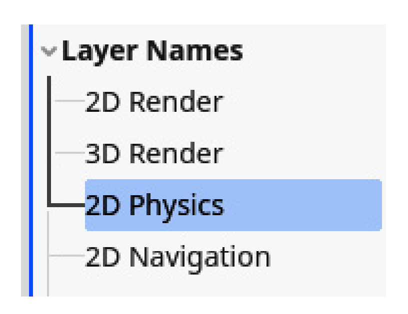


Рисунок 9.29 – Именование слоев столкновений в категории 2D Физика

Здесь вы можете увидеть различные слои столкновений и их названия. Ни у одного из них пока нет названия.

1. Дайте слою **Layer 1** имя **Collision** и слою **Layer 2** имя **Collectible**:

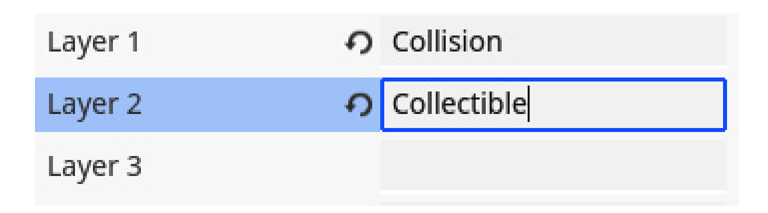


Рисунок 9.30 – Наименование двух слоёв

1. Если теперь снова выбрать узел **Area2D** в сцене collectible.tscn и навести курсор на номера слоев, мы увидим всплывающее имя:

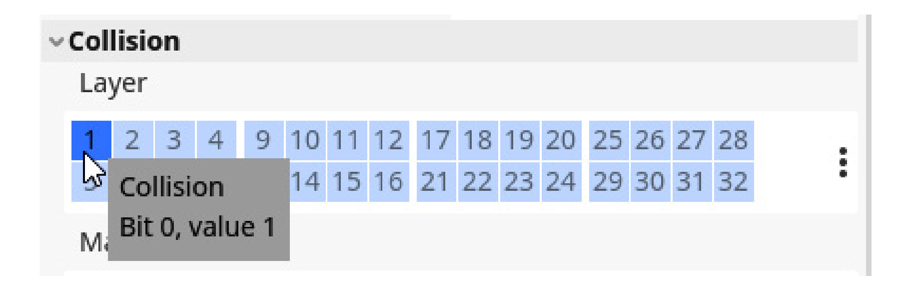


Рисунок 9.31 – При наведении курсора на номер слоя столкновений отображается его название

1. Мы также можем нажать на многоточие рядом со слоями для более удобного выбора слоёв и увидеть там данные нами имена.

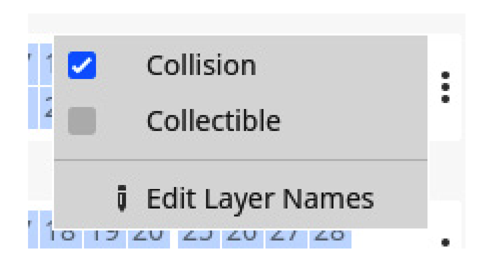


Рисунок 9.32 – Нажатие на многоточие упрощает выбор именованных слоёв столкновений

С этими слоями столкновений, имеющими имя, будет проще назначать их в будущем. Давайте сделаем это в следующем разделе.

Назначение правильных слоёв

Теперь, когда мы разобрались со слоями и масками столкновений и знаем, как их называть, давайте настроим их так, чтобы только игрок мог активировать коллекционные предметы.

Нам придется настроить слои столкновений и маски всех физических тел в игре. К счастью, мы сделали отдельные сцены для всех из них, так что это будет происходить быстро, и в будущем мы сможем учитывать эти слои при создании сцен.

Для корневого узла **player.tscn** настройте слои и маску следующим образом:

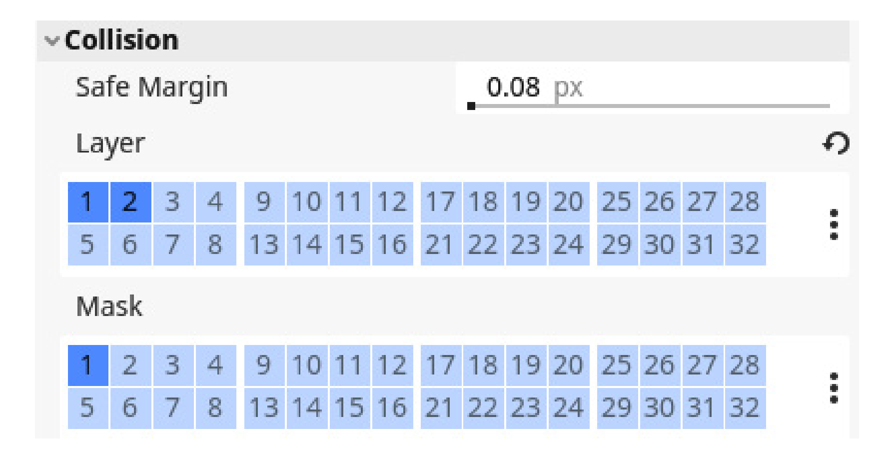


Рисунок 9.33 – Конфигурация слоя столкновений и маски для персонажа игрока

Для **boulder.tscn** и **wall.tscn** нам нужна следующая конфигурация:

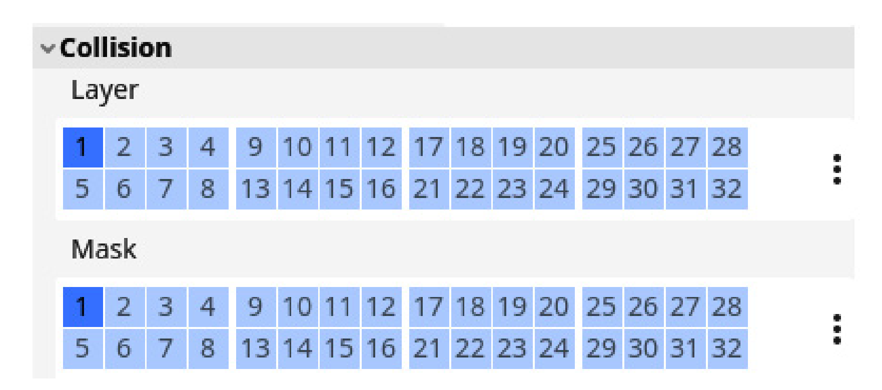


Рисунок 9.34 – Конфигурация слоя столкновений и маски для валунов (boulders) и стен (walls)

Наконец, для сцены **collectible.tscn** установите следующую конфигурацию:

Рисунок 9.35 – Конфигурация слоя столкновения и маски для предметов коллекционирования

Вы можете видеть, что игрок, валуны и стена находятся как в слое столкновений, так и в маске столкновений. Это потому, что они должны иметь возможность взаимодействовать друг с другом. С другой стороны, игрок находится в слое предметов коллекционирования, а не в маске предметов коллекционирования, в то время как сцена предметов коллекционирования делает наоборот. Мы определяем слой и маску для предметов коллекционирования таким образом, потому что игроку не нужно напрямую взаимодействовать с предметами коллекционирования и не нужно их обнаруживать; сцена предметов коллекционирования делает всю эту работу за нас.

Ваш ход!

Отлично, мы создали наше зелье здоровья! Теперь вы можете реализовать монету, чтобы игрок мог собирать золото. Вот некоторые шаги, которые вы можете предпринять:

1. Создайте новую унаследованную сцену из сцены **collectible.tscn** , как мы видели в разделе *Наследование от базовой сцены*.
2. Расширьте скрипт collectible.gd, как мы это делали в разделе *Написание кода для предметов коллекционирования*.
3. Отслеживайте количество золота, которым владеет игрок,

   с помощью переменной.
4. Покажите на экране, сколько монет у игрока, с помощью узла метки Label.

Возможные реализации всего этого я оставлю в репозитории проекта.

Мы многому научились в этом разделе. Мы узнали, что такое узлы **Area2D**, а слои столкновений и маски больше не являются тайной, а полезным инструментом для определения того, с какими телами и областями мы хотим взаимодействовать. Давайте сделаем несколько последних упражнений, прежде чем подвести итоги и закончить главу.

Дополнительные упражнения – Заточка топора

1. Мы добавили столкновения к валунам и внутренним стенам арены, но не к внешним стенам. Добавьте **StaticBody2D**, который не даст игроку сбежать с арены.
2. Создайте базовую сцену для валунов, унаследуйте два валуна из неё и сделайте их формы разными. Также убедитесь, что вы обновили форму столкновения, чтобы игрок правильно с ними сталкивался.

Итоги

Мы начали эту главу с изучения узла **Camera2D** и того, как сделать его более приятным и удобным для игрока, чтобы ему не приходилось думать о нём во время навигации по игровому полю.

После этого мы добавили коллайдеры игроку и всем твердым объектам в игре. Мы даже использовали формы столкновений для создания небольших коллекционных предметов, таких как зелье здоровья.

Попутно мы увидели, что такое сигналы и как их можно связать с функциями в скрипте узла.

В следующей главе мы дополним нашу игру врагами и меню, чтобы у нас получился полноценный игровой цикл.

Опрос

- Почему мы использовали точку перед игроком для позиционирования камеры?
- Что представляет собой последний параметр функции **lerp Vector2**? Вот пример:

```
var position: Vector2 = Vector2(1, 1)
var target_position: Vector2 = Vector2(3, 5)
position.lerp(target_position, 0.5)
```

- Почему мы использовали **CharacterBody2D** для персонажа игрока, а не **RigidBody2D**?
- Для чего используются узлы **Area2D**?
- У нас есть два объекта: узел **Area2D** и узел **CharacterBody2D**. Мы хотим иметь возможность обнаружить **CharacterBody2D** с помощью узла **Area2D**. Как нам нужно настроить их слои столкновений и маски?
- Узлы **Area2D** и **CharacterBody2D** должны находиться в одном слое столкновений.
- Узел **Area2D** должен находиться в той же маске столкновений, что и слой столкновений узла **CharacterBody2D**.
- Узлы **Area2D** и **CharacterBody2D** должны находиться в одной и той же маске столкновений.
- Сигналы уведомляют нас об определённых действиях, которые происходят в узле. К какому сигналу мы подключились, чтобы определить, вошёл ли игрок в узел **Area2D** коллекционного предмета?

10

Создание меню и врагов, использование автозагрузок

Хотя было очень весело настраивать все текущие системы, игра всё ещё немного скучна. Нет настоящего противника, ничего, что могло бы помешать игроку просто собрать все желаемые золотые монеты. Давайте добавим немного сложности, создав врагов, которые нападают на игрока и пытаются остановить его на пути к славе и известности!

Далее мы также создадим небольшое меню, чтобы начать нашу игру. Мы сделаем это с помощью системы **пользовательского интерфейса (user interface)** Godot, сокращённо — **UI**, которая использует узлы управления — **Control**. В этой главе мы обсудим следующие темы:

- Создание меню
- Создание врагов
- Стрельба снарядами
- Набор очков в автозагрузках

Технические требования

Как и для каждой предыдущей главы, окончательный код можно найти в репозитории GitHub в подпапке для этой главы: https://github.com/PacktPublishing/Learning-GDScript-by-Developing-a-Game-with-Godot-4/tree/main/chapter10.

Создание меню

Самая захватывающая часть разработки игры — это, конечно,

создание самой игры! Заставить вещи двигаться, сражаться, прыгать, стрелять, взаимодействовать и так далее. Но есть и другая часть, которая не менее важна: пользовательский интерфейс. Пользовательский интерфейс связывает всё воедино. Он информирует игрока о том, что происходит, и позволяет ему легко перемещаться из меню в меню, не думая о том, как перейти из одного интерфейса в другой.

Хороший пользовательский опыт, пользовательский интерфейс или проектирование взаимодействия человека с компьютером — это сложно! Но всё начинается с изучения того, как сделать пользовательский интерфейс в первую очередь. Итак, давайте посмотрим, как мы можем создавать меню и интерфейсы.

Узлы управления Control

Движок Godot поставляется с обширной библиотекой узлов управления. Мы уже использовали один из них, узел метки — **Label**, в *Главе 6*. Эти узлы называются **узлами Control** и помечаются зелёным цветом:

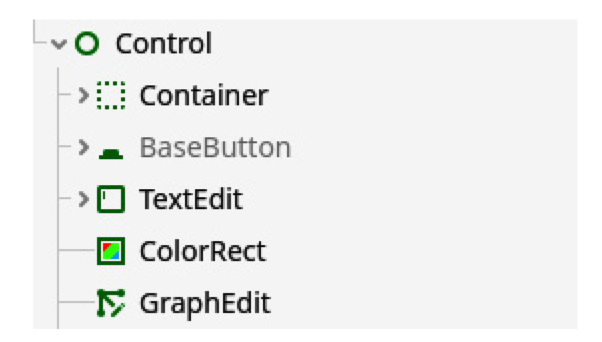


Рисунок 10.1 – Некоторые узлы управления можно распознать по их зелёному цвету

Если вы перейдёте в меню **Create New Node** и откроете его, вы увидите, что есть много таких узлов **Control**. Мы можем разделить их на три разные группы. Давайте рассмотрим некоторые узлы в каждой группе, и то, что они могут сделать для нас.

Узлы, показывающие информацию

Первая группа узлов показывает информацию. В этой группе вы найдёте узел **Label**, который мы использовали в *Главе 6*, а также узлы **ColorRect** и **TextureRect**:

- **Label**: отображает короткую строку текста.
- **RichTextLabel**: отображает более длинный фрагмент текста, который можно отформатировать определённым образом.
- **ColorRect**: отображает сплошной прямоугольник одного цвета.
- **TextureRect**: Показывает текстуру в прямоугольнике. Этот узел похож на узел **Sprite2D** — они оба используются для отображения текстуры, но в разных контекстах.

На следующем рисунке вы можете увидеть, как эти узлы выглядят в редакторе:

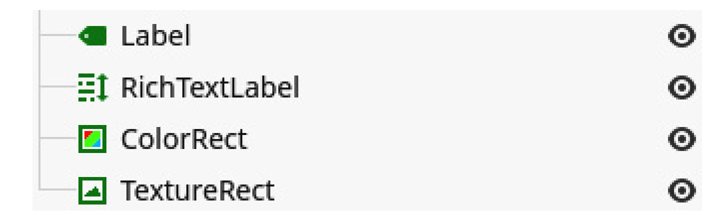


Рисунок 10.2 – Примеры узлов управления, которые отображают информацию

Все эти узлы что-то показывают пользователю.

Узлы, принимающие ввод

Любой хороший UI также может принимать ввод, например, с помощью кнопок. Вот некоторые из узлов ввода, которые предоставляют узлы UI движка Godot:

- **Button**: простая кнопка, на которую можно нажать.
- **CheckBox**: флажок, который можно ставить и снимать.
- **CheckButton**: то же самое, что и флажок, но с другим

внешним видом.

- **LineEdit**: простой узел, который может принимать одну строку текстового ввода и предоставлять её в виде строки.
- **HSlider** и **VSlider**: ползунки, которые используются для ввода числа. **HSlider** скользит горизонтально, а **VSlider** — вертикально.

На следующем рисунке вы можете увидеть, как эти узлы выглядят в редакторе:

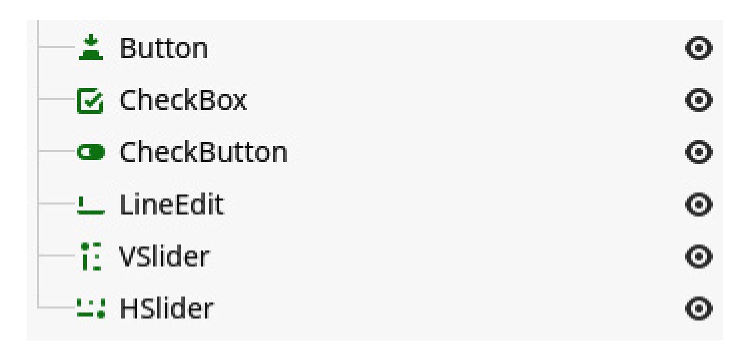


Рисунок 10.3 – Примеры узлов управления, которые принимают ввод

Все эти узлы тем или иным образом принимают ввод.

Узлы, содержащие другие узлы

Наконец, есть узлы, которые вы не видите, но которые очень важны, потому что они обеспечивают правильное размещение всех остальных элементов пользовательского интерфейса. Эти узлы формируют скелет, в котором другие узлы управления **(Control)** могут найти своё место.

Узлы-контейнеры **(Container)** помогают нам составлять пользовательский интерфейс так, как мы хотим. Этот тип узла может отображать элементы рядом друг с другом, добавлять некоторое расстояние между узлами и т.д.

Эти контейнеры также могут помочь сохранить интерфейс удобным и красивым при изменении размера экрана. Это

случается нечасто, но в наши дни в игры можно играть на экранах самых разных размеров и соотношений сторон. Просто подумайте о разнице между экраном компьютера и экраном телефона.

Вот некоторые интересные узлы-контейнеры:

- **VBoxContainer** и **HBoxContainer**: аккуратно упорядочивают все дочерние узлы по вертикали или горизонтали.
- **CenterContainer**: Центрирует дочерние узлы.
- **GridContainer**: Оорганизует все свои дочерние узлы в аккуратную сетку.
- **MarginContainer**: добавляет пространство вокруг дочерних узлов.
- **Panel**: обеспечивает фон, показывающий логическую принадлежность этой части пользовательского интерфейса к единому целому.

На следующем рисунке вы можете увидеть, как эти узлы выглядят в редакторе:

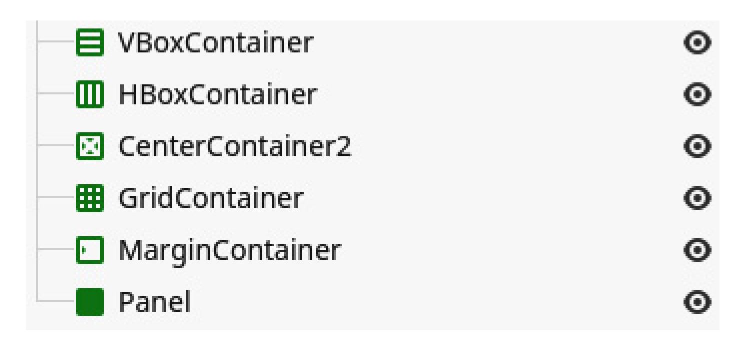


Рисунок 10.4 – Примеры узлов управления, которые могут содержать другие узлы

Все узлы-контейнеры **(Container)** размещают содержащтеся в них дочерние узлы определённым образом.

Списки узлов в этом разделе не являются исчерпывающими. Быстрый взгляд на категорию узлов **Control** при добавлении узла делает это довольно очевидным. Но это самые важные

узлы, которые вы, скорее всего, будете использовать в первую очередь. Остальные более специализированы.

Самое интересное, что весь редактор Godot создан из этих узлов управления, просто чтобы показать, насколько они гибки и эффективны для создания пользовательских интерфейсов.

Теперь, когда у нас есть базовые знания о различных узлах **Control**, мы можем приступить к созданию меню с их помощью.

Создание базового меню запуска

Давайте создадим меню запуска, которое будет отображаться при старте игры. Это меню должно просто отображать название игры, кнопку для начала игры, кнопку для выхода из игры, и, например, мы могли бы добавить некоторую информацию о том, кто создал игру:



Рисунок 10.5 – Вот как будет выглядеть наше меню запуска

Давайте рассмотрим шаги по созданию меню запуска, которое показано на *Рисунке 10.5*:

1. Создайте новую сцену с именем **menu.tscn** в новой папке **screens/ui**.
2. Выберите **Пользовательский интерфейс (User Interface)**

в качестве типа корневого узла:

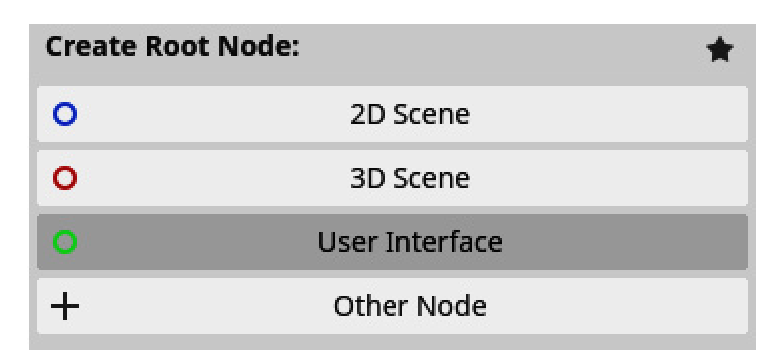


Рисунок 10.6 – Выбор узла пользовательского интерфейса в качестве корневого узла для нашего меню

1. Переименуйте корневой узел в **Menu**.
2. Начнём с добавления в меню узла **ColorRect**; это будет наш фоновый цвет.
3. Теперь, чтобы растянуть узел **ColorRect** на весь экран, используем удобное маленькое меню в верхней панели, которое имеют все узлы **Control**. Выберите узел **ColorRect** в дереве сцены и выберите **Full Rect** из списка **Пресет якорей (Anchor preset)**:

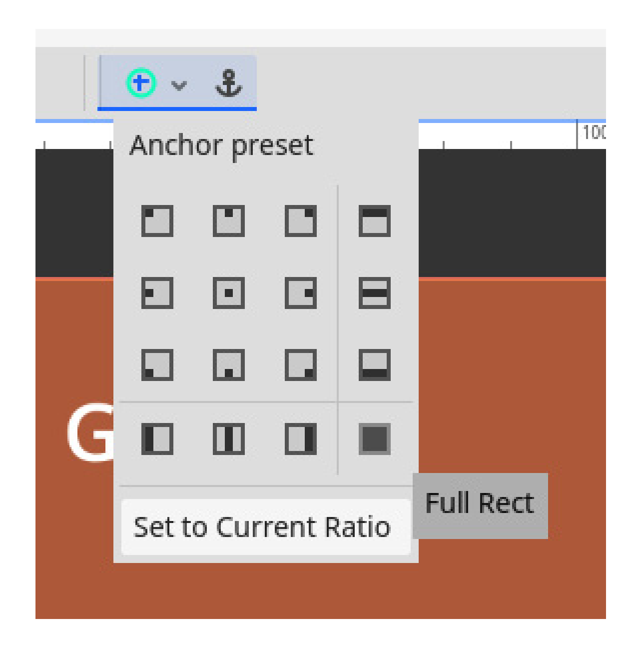


Рисунок 10.7 – Выбор Full Rect для того, чтобы узел ColorRect покрывал весь экран

1. Теперь добавьте узел **CenterContainer** к корневому узлу, сделайте его дочерним узлом **VBoxContainer** и назовите его **MainUIContainer**.
2. Теперь добавьте узел **Label** как первый дочерний элемент под узлом **MainUIContainer**. Переименуйте его в **TitleLabel**. Этот узел метки будет отображать название нашей игры:

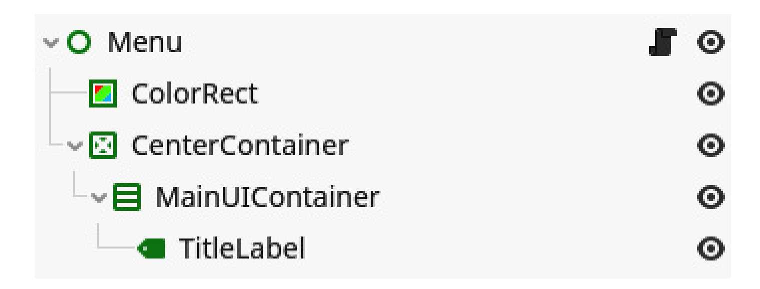


Рисунок 10.8 – Дерево сцен нашего меню на настоящий момент

1. Придумайте хорошее название для игры и введите его в поле **Text** узла **TitleLabel**.
2. Теперь перейдите в раздел переопределения темы — **Theme Overrides** и установите для параметра **Размер шрифта (Font Size)** значение, более подходящее для названия игры, например, **60** пикселей:

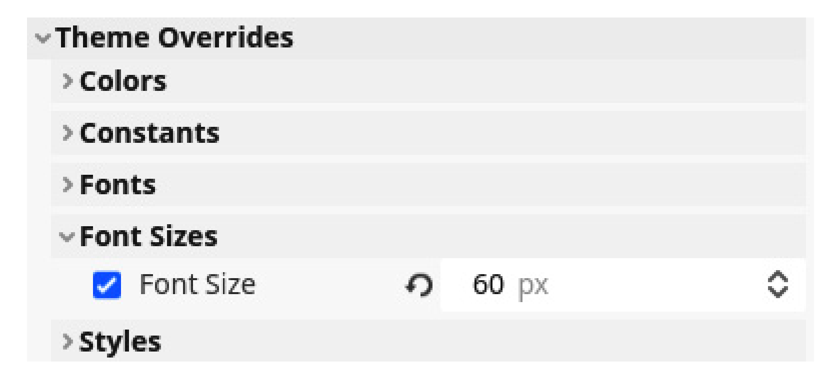


Рисунок 10.9 – Вы можете изменить размер шрифта метки через переопределения темы — Theme Overrides

Это было сделано только для создания заголовка для нашего игрового пользовательского интерфейса. Может показаться, что шагов много, но некоторые из узлов, которые мы использовали, позволят легко расширить пользовательский интерфейс (UI) на следующих нескольких шагах.

Добавим панель с кнопками и строкой с именем автора:

1. Добавьте узел **PanelContainer** к узлу **MainUIContainer**.
2. Теперь создайте в этом контейнере панели следующую структуру:

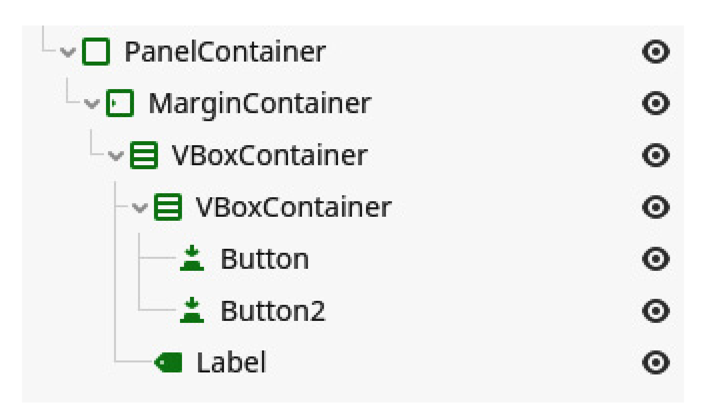


Рисунок 10.10 – Структура дерева сцены для узла PanelContainer и его содержимого

1. Переименуйте первую кнопку в **PlayButton** и измените её текст на **ИГРАТЬ (PLAY)**.
2. Переименуйте вторую кнопку в **ExitButton** и измените её текст на **ВЫЙТИ (EXIT)**.
3. Переименуйте узел метки Label в **CreditLabel** и измените её текст на любой другой!
4. Теперь перейдите в первый узел **VBoxContainer** и установите значение константы **Separation** на **50** пикселей.
5. Установите значение константы **Separation** второго узла **VBoxContainer** на **20** пикселей.
6. Наконец, установите значение константы **Separation** узла **MainUIContainer** на **200** пикселей.

Отличная работа — макет UI готов. Полное дерево сцены должно выглядеть так:

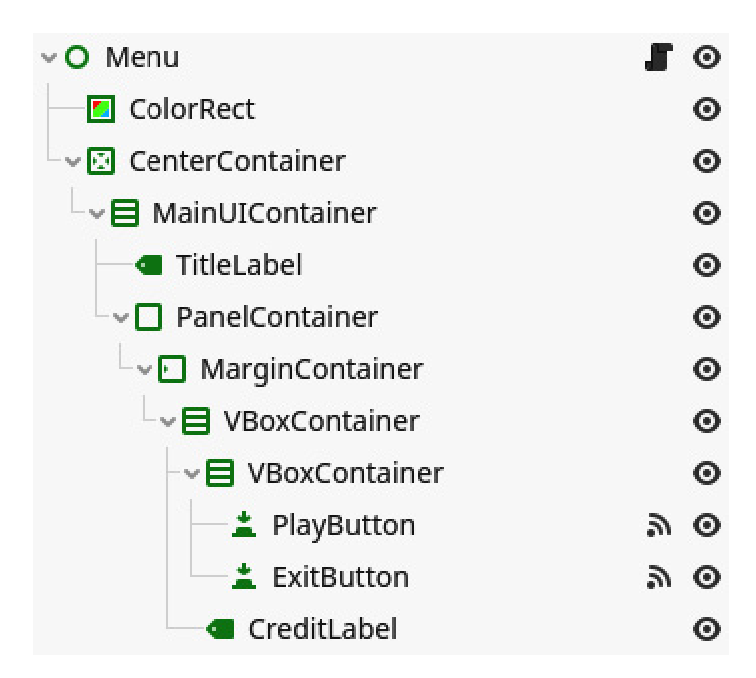


Рисунок 10.11 – Полное дерево сцены нашего меню

Осталось только сделать кнопки функциональными! Давайте сделаем это очень быстро:

1. Добавьте пустой скрипт в корневой узел **Menu** с именем **menu.gd**.
2. Теперь подключите сигнал нажатия узла **PlayButton** к этому узлу.

   Тело подключённой функции довольно простое:

   ```
   func _on_play_button_pressed():
       get_tree().change_scene_to_file("res://screens/g
   ```

3. Так же подключите сигнал нажатия узла **ExitButton**.

   Тело этой функции ещё проще:

```
func _on_exit_button_pressed():
   get_tree().quit()
```

В предыдущих фрагментах кода мы обращались к корню дерева сцены с помощью **get_tree()**. Эта функция возвращает **SceneTree**, объект, который управляет всей иерархией узлов во время работы игры.

В функции, которая связана с кнопкой **Play**, мы вызываем функцию **change_scene_to_file()** на этом объекте, которая переключает текущую запущенную сцену на ту, которая указана в пути, который мы предоставляем функции. Таким образом, чтобы запустить основную игровую сцену, мы просто указываем ей путь, начиная с корня проекта, к сцене **main.tscn**.

Важное примечание

Стоит отметить, что в момент вызова **change_scene_to file()** начнёт загружать файл сцены, на который предполагается переключиться. Это означает, что игра заблокируется или зависнет на время загрузки. Это не очень хорошо, когда мы переключаемся на большую сцену, чего мы, к счастью, не делаем в нашем случае.

В функции, которая связана с кнопкой **Exit**, мы вызываем функцию **quit()**, которая просто останавливает среду выполнения.

Теперь вы можете опробовать меню, запустив его!

Установка главной сцены

Чтобы убедиться, что наше меню является главной сценой, которая загружает игру, нам нужно быстро перейти в настройки проекта, чтобы объявить это. В настройках проекта в разделе **Приложение | Запустить (Application | Run)** укажите, что **menu.tscn** является главной сценой:

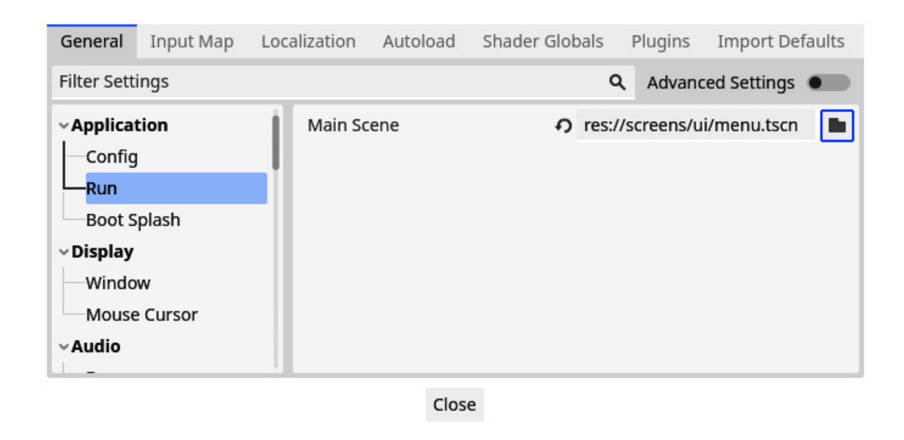


Рисунок 10.12 – Настройка главной сцены в настройках проекта

Это гарантирует, что при запуске игры с помощью кнопки **Запустить проект (Play)** вверху или с помощью сочетания клавиш *F5* сцена **menu.tscn** будет запущена по умолчанию.

Важное примечание

Помните, что когда основная сцена не установлена и мы запускаем игру вышеупомянутыми методами, Godot спросит нас, хотим ли мы использовать текущую открытую сцену в качестве основной.

Мы узнали много нового о узлах управления **Control** и о том, как использовать их для быстрого создания UI. Давайте пойдем и создадим врагов.

Создание врагов

В реальной жизни создание врагов не является хорошей идеей. Но в контексте разработки видеоигр это часто является отличным способом бросить вызов игроку и столкнуть его с некоторым сопротивлением.

Враг, которого мы будем создавать, довольно прост и понятен. Но мы все равно многому научимся по пути — например, как позволить врагам перемещаться к игроку, чтобы атаковать его.

Как я уже сказал, мы сделаем врага простым. Мы сделаем врага, который появляется в случайное время в случайном месте арены и начинает атаковать игрока. С того момента, как враг касается игрока, мы вычитаем одно очко здоровья из жизни игрока и удаляем врага из игры. Таким образом, у игрока есть противники, но он будет подавлен ордой врагов.

В следующем разделе, *Стрельба снарядами*, мы разработаем способ, которым игрок может защитить себя. Но сейчас мы сосредоточимся исключительно на враге и его поведении.

Построение базовой сцены

Как и в любой новой части нашей игры, начнём с создания базовой структуры в дереве сцены для врага и добавим код и другие интересные вещи позже в этом разделе:

1. Создайте папку **parts/enemy** и внутри неё создайте новую сцену с именем **enemy.tscn**.
2. Воссоздайте следующее дерево сцены. Обратите внимание, что корнем является узел **CharacterBody**:



Рисунок 10.13 – Дерево сцены нашего врага (Enemy)

1. Выберите спрайт из папки **assets/sprites/enemies** в качестве текстуры для узла **Sprite2D**:

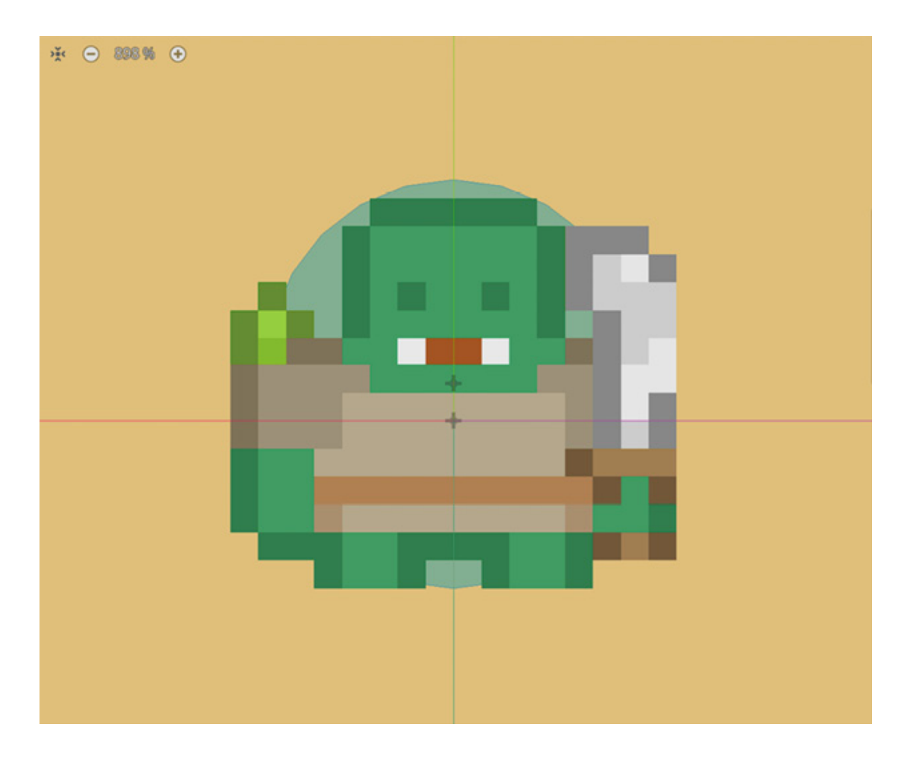


Рисунок 10.14 – Сцена врага, Enemy, в 2D-редакторе

1. Убедитесь, что вы установили масштаб (Scale) узла спрайта на **(3, 3)**.

На данный момент сцена **Enemy** очень проста. Давайте посмотрим, как мы могли бы сделать навигацию для них, чтобы немного её усложнить.

Навигация для врагов

Мы можем легко заставить врагов двигаться прямо к игроку. Проблема в том, что они будут застревать в стенах и натыкаться на валуны, что не выглядит очень естественно и заставляет их выглядеть довольно глупо.

К счастью, движок Godot имеет свойство **NavigationServer**, которое вычисляет путь вокруг всех этих препятствий и делает движение врагов более естественным и плавным.

Для этого мы рассмотрим два новых узла: **NavigationRegion2D** и **NavigationAgent2D**.

Создание узла NavigationRegion2D

Во-первых, нам нужно определить, в какой области уровня может перемещаться наш враг, затем мы хотим вырезать из этой области места, где находится стена или валун. Это именно то, что делает узел **NavigationRegion2D**! Давайте определим один:

1. Перейдите на игровую сцену **main.tscn**.
2. В корневом узле с именем **Main** добавьте узел **NavigationRegion2D**.
3. Щёлкните пустое свойство **Navigation Polygon** и выберите **New NavigationPolygon**:

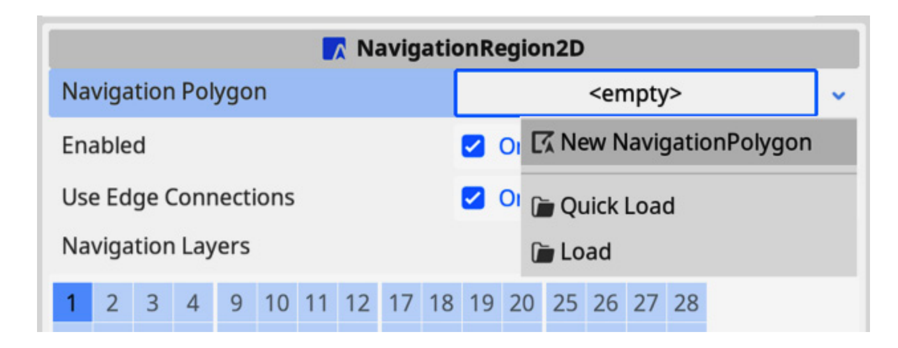


Рисунок 10.15 – Нажатие на New NavigationPolygon

1. Теперь мы сначала определим внешние границы, где враги смогут двигаться. Нарисуйте многоугольную форму, щёлкнув в редакторе. Постарайтесь тщательно обвести внешнюю часть арены. Не забудьте замкнуть форму, щёлкнув первую размещенную вами точку:

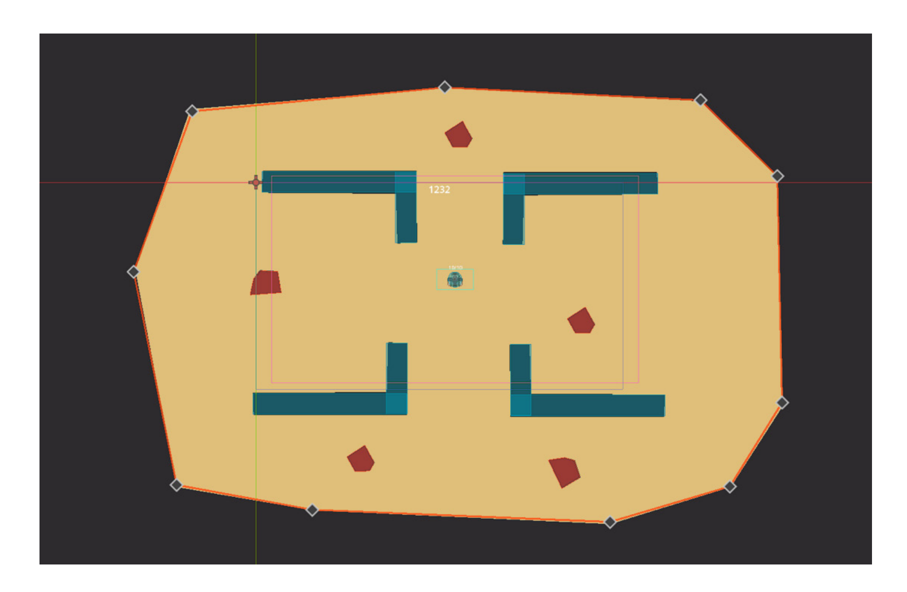


Рисунок 10.16 – Создание внешних границ узла NavigationRegion2D

1. Нажмите **Запечь NavigationPolygon (Bake NavigationPolygon)** в верхней части окна, чтобы создать навигационный полигон:

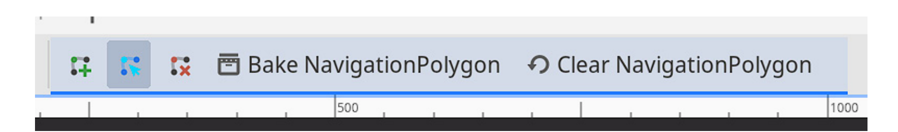


Рисунок 10.17 – Нажатие Bake NavigationPolygon

После выполнения этих шагов узел **NavigationRegion2D** должен выглядеть следующим образом:

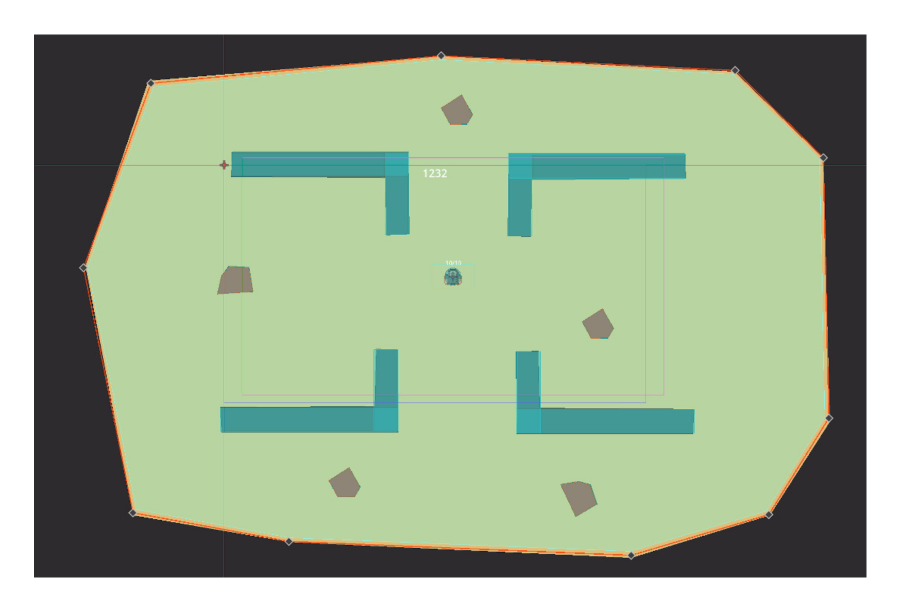


Рисунок 10.18 – Узел NavigationRegion2D после запекания полигона в первый раз

Сине-зеленоватая область — это то место, где враги смогут перемещаться и двигаться. Но вы уже видите проблему – эта область также простирается над нашими стенами и валунами. Мы не хотим, чтобы враги думали, что они могут проходить сквозь них, потому что, ну, они не могут; это статические физические тела. К счастью, у Godot есть функционал для автоматического обнаружения этих объектов и запекания свойства **NavigationPolygon** таким образом, чтобы оно учитывало их.

Разверните свойство **Navigation Polygon** узла **NavigationRegion2D**, щёлкнув по нему, и настройте его следующим образом:

1. Установите **Geometry | Parsed Geometry Type** на **Static Colliders**. Мы делаем это, чтобы учитывать только статические коллайдеры при автоматической генерации.
2. Установите **Geometry | Source Geometry Mode** на **Group With Children**. Таким образом, автоматическая генерация будет сканировать дочерние элементы узлов, чтобы найти статические коллайдеры.

3. Установите **Agents | Radius** на **40** пикселей. Эта настройка задаёт радиус агентов, которых хотим использовать в узле **NavigationRegion2D**, и автоматическая генерация может это учитывать, чтобы агенты не сталкивались с препятствиями, которых они должны избегать:

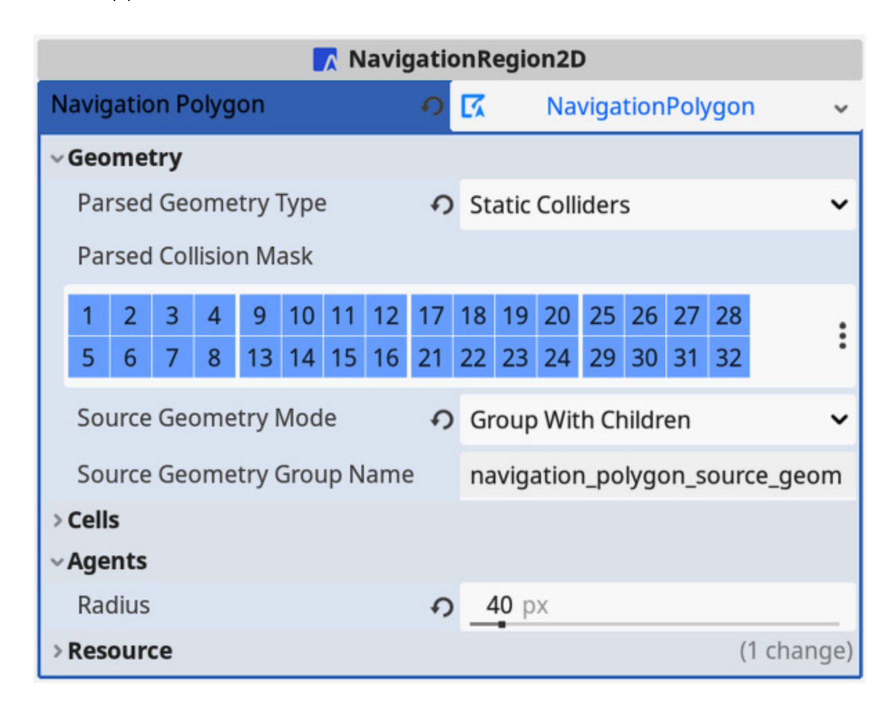


Рисунок 10.19 – Настройка свойства NavigationPolygon

1. Выберите узел **Arena** и переключитесь на вкладку **Узел (Node)**, расположенную рядом с вкладкой **Инспектор (Inspector)**:

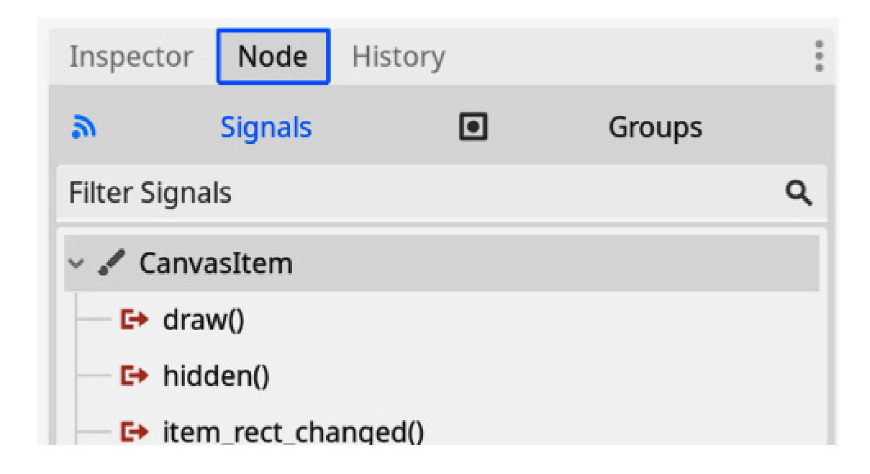


Рисунок 10.20 – Переход на вкладку Узел

1. Перейдите на вкладку **Группы (Groups)**, которая находится рядом с вкладкой **Сигналы (Signals)**:

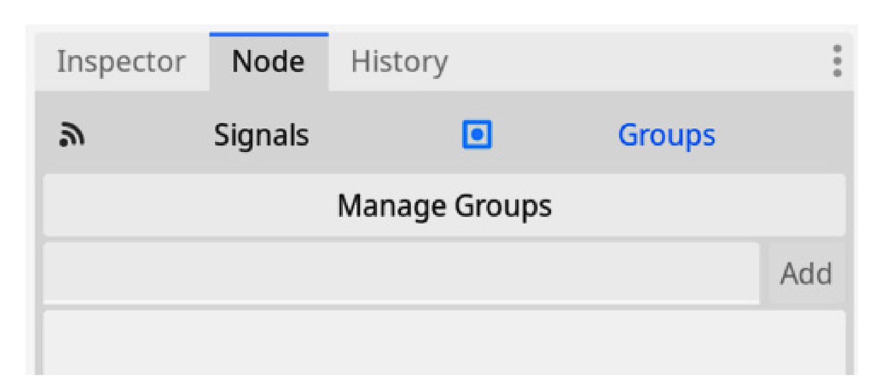


Рисунок 10.21 – Переход на вкладку Группы

1. Вставьте **navigation_polygon_source_geometry_group** в текстовое поле и нажмите **Добавить (Add)**:

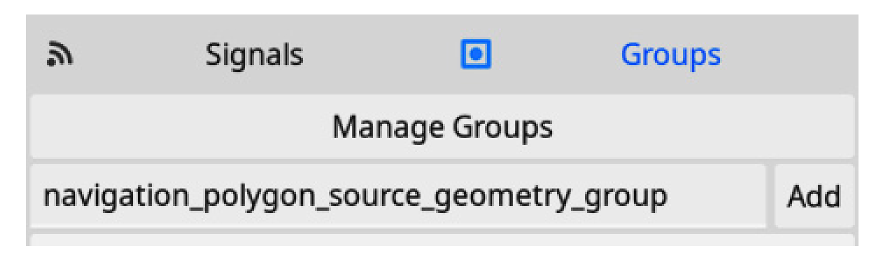


Рисунок 10.22 – Добавление группы navigation_polygon_source_geometry_group

1. Теперь снова выберите узел **NavigationRegion2D** и снова нажмите **Bake NavigationPolygon**.

Когда вы закончите, область навигации должна выглядеть так:

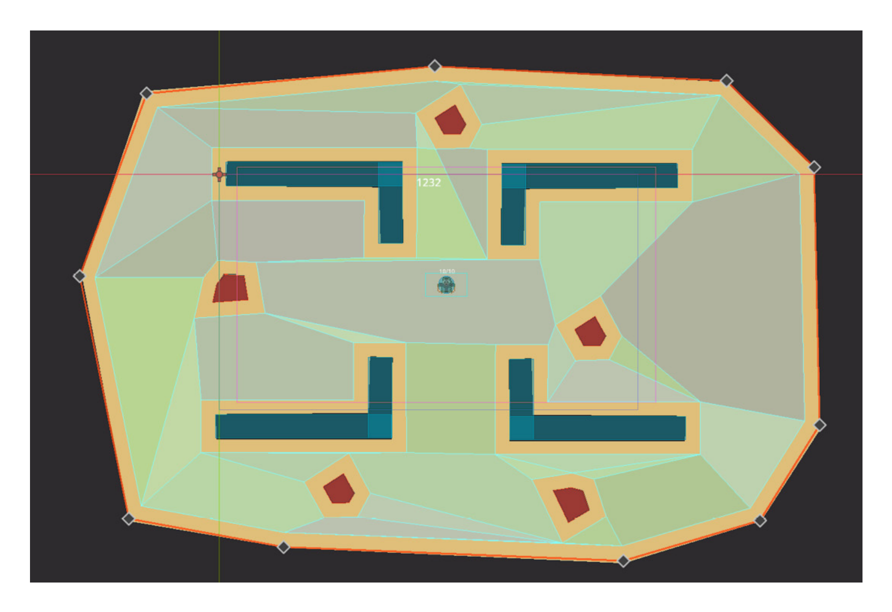


Рисунок 10.23 – Результирующий узел NavigationRegion2D

Сине-зеленоватая область теперь прекрасно обходит стены и валуны. Вы также можете видеть, что есть некоторый зазор между препятствиями и тем местом, где начинается область. Это то, что мы настроили при определении свойства **Radius** агентов. Этот зазор гарантирует, что поиск пути не будет слишком близко к препятствиям, заставляя врагов избегать столкновений с ними.

На *шагах 2—4* мы добавили узел **Arena** в **группу узлов**. Мы сделали это, потому что при запекании свойства **NavigationPolygon** оно будет искать все узлы в группе узлов **navigation_polygon_source_geometry_group** и учитывать статические тела внутри них. Давайте сделаем небольшое отступление, чтобы поговорить о группах узлов.

Что такое группы узлов?

Группы или **группы узлов** в движке Godot подобны тегам в других программах. Вы можете добавить любое количество групп к узлу. Мы можем просто сделать это через вкладку **Группы**, как мы делали в шагах предыдущего раздела.

Группы чрезвычайно полезны, поскольку вы можете, например, делать следующее:

- Проверить, является ли узел частью группы.
- Получить все узлы внутри группы из дерева.
- Вызывать методы на всех узлах внутри группы.

Мы ещё будем использовать группы позднее.

Узел **NavigationRegion2D** готов, теперь давайте рассмотрим процесс добавления узла **NavigationAgent2D** к сцене **Enemy**.

Добавление узла NavigationAgent2D к сцене Enemy

Последнее, чем нам нужно дополнить сцену **enemy.tscn** в редакторе, прежде чем начать писать код, — это узел **NavigationAgent2D**. Этот узел управляет поиском пути и навигацией в узле **NavigationRegion2D** , который мы создали в предыдущем разделе.

Просто добавьте узел **NavigationAgent2D** Просто добавьте узел **Enemy**. Нам не нужно делать никаких других настроек:

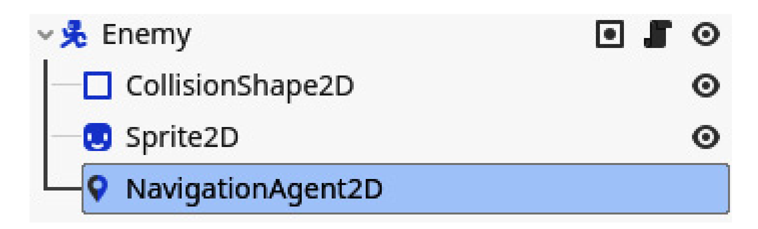


Рисунок 10.24 – Добавление узла NavigationAgent2D в сцену Enemy

Теперь мы можем начать писать код для нашего врага!

Написание скрипта для врагов

Код для нашего врага будет очень похож на код для нашего игрока. Они оба перемещаются на основе физики ускоряющейся скорости (accelerating velocity). Единственное отличие в том, что для врага позиция, куда он хочет переместиться, определяется свойством **NavigationServer**. Этот сервер смотрит на узел **NavigationRegion2D** и текущее положение узла **NavigationAgent2D** чтобы вычислить наилучший маршрут к точке на карте, которую мы выбираем для перехода.

Давайте начнём с написания шаблонного кода, определяющего некоторые движения нашего врага:

```
class_name Enemy extends CharacterBody2D
@onready var _navigation_agent_2d: NavigationAgent2D = $
@export var max_speed: float = 400.0
@export var acceleration: float = 1500.0
@export var deceleration: float = 1500.0
var player: Player
func _physics_process(delta: float):
   _navigation_agent_2d.target_position = player.global_
   if _navigation_agent_2d.is_navigation_finished():
      velocity = velocity.move_toward(Vector2.ZERO, dece
   else:
      var next_position: Vector2 = _navigation_agent_2d.
      var direction_to_next_position: Vector2 = global_p
      velocity = velocity.move_toward(direction_to_next_
   move_and_slide()
```

В целом этот код очень похож на код движения, который мы написали для скрипта **player.gd**. Единственное отличие в том, что теперь мы используем узел **NavigationAgent2D**, чтобы указать, куда нам нужно идти:

```
_navigation_agent_2d.target_position = target.global_pos
```

Как вы видите, мы идём к глобальной позиции переменной **player**. Мы определим эту переменную **player** немного позже.

Переменные position и global_position

Переменная **position** узла **Node2D** является **всегда** позицией относительно его родительского узла. Переменная **global_position**, с другой стороны, является позицией узла в мировом пространстве относительно корня дерева сцены. Обе автоматически обновляются, когда узел перемещается в 2D-пространстве; это по сути те же данные, но с другой точкой отсчета.

Здесь нам необходимо использовать переменную **global_position**, поскольку целевая позиция узла **NavigationAgent2D** должна быть глобальной позицией.

Затем нам нужно проверить, нужно ли нам перемещаться или нет:

```
if _navigation_agent_2d.is_navigation_finished():
```

Если нам нужно переместиться, мы спрашиваем узел **NavigationAgent2D**, в какую следующую позицию нам следует переместиться:

```
var next_position: Vector2 = _navigation_agent_2d.get_ne
```

Затем все, что нам нужно сделать, это вычислить направление от нашей текущей позиции до этой следующей позиции, а остальная часть кода точно такая же, как для сцены **Player** из *Главы 7*.

Чтобы выбрать узел **Player**, мы воспользуемся группами узлов, добавив эту функцию **_ready()**:

```
func _ready():
	var player_nodes: Array = get_tree().get_nodes_in_grc
	if not player_nodes.is_empty():
```

```
    target = player_nodes[0]
```

Чтобы получить узел player из дерева сцены, мы сделаем что-то новое. Мы запросим у текущего дерева сцены все узлы, которые находятся в группе **player**. Эта функция вернёт массив с узлами, которые принадлежат этой группе. Поэтому нам придется взять первый элемент, если он есть.

Важное примечание

Может показаться странным запрашивать все узлы player в сцене, когда есть только один. Мы делаем это, чтобы можно было использовать примерно тот же код для нацеливания на большее количество игроков, когда мы будем иметь дело с несколькими игроками в следующей главе.

Группы узлов являются полезной функцией движка Godot, поскольку движок отслеживает все узлы в группе, чтобы мы могли легко запрашивать их или проверять, принадлежит ли узел определенной группе.

Этот код пока не будет работать, потому что узел player на самом деле ещё не в группе **player**! Чтобы добавить его в эту группу, нам нужно немного изменить сцену **Player**:

1. Перейдите в сцену **player.tscn**.
2. Выберите корневой узел.
3. В окне, содержащем сигналы узла, есть кнопка **Группы (Groups)**. Нажмите её, и вы увидите окно **Группы (Groups)**:

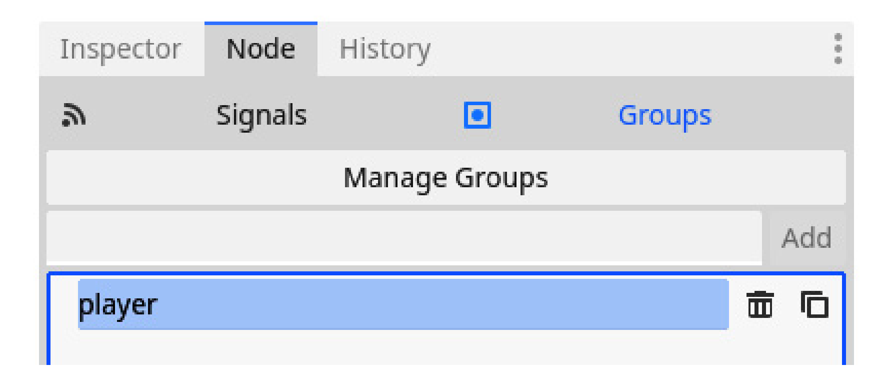


Рисунок 10.25 – Добавление корневого узла player в группу узлов с именем player

1. Здесь введите **player** в строку ввода и нажмите **Добавить (Add)**.

Поместите врага в главную сцену (main), и вы увидите, что он начнёт двигаться к игроку! Это здорово. Но враги должны иметь возможность наносить урон игроку. Этим мы и займёмся далее.

Нанесение урона игроку при столкновении

Чтобы определить, находится ли враг достаточно близко к игроку, чтобы нанести урон, мы будем использовать узел **Area2D**, как мы это делали для предметов коллекционирования в *Главе 9*:

1. Давайте начнём с добавления функции **get_hit()** в скрипт **player.gd**. Эта функция будет вызываться, когда игрок получает удар от врага и снижает здоровье игрока — health:

```
func hit():
   health -= 1
```

2. Добавьте узел **Area2D** к сцене **enemy.tscn** и назовите его **PlayerDetectionArea**.
3. Под этой областью добавьте узел **CollisionShape2D**:

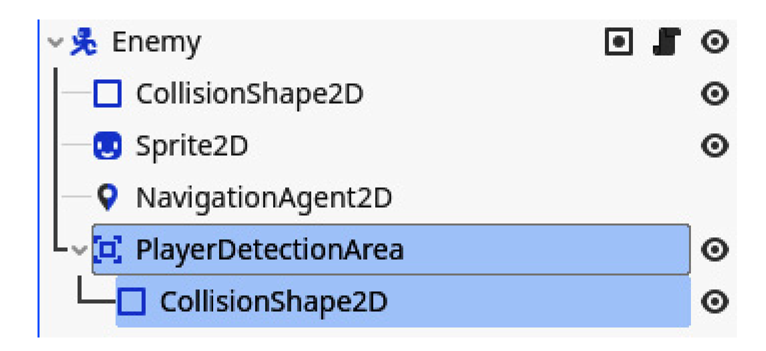


Рисунок 10.26 – Добавление узла Area2D к сцене Enemy

1. Придайте этому узлу столкновения форму **CircleShape2D**, которая будет немного больше спрайта врага:

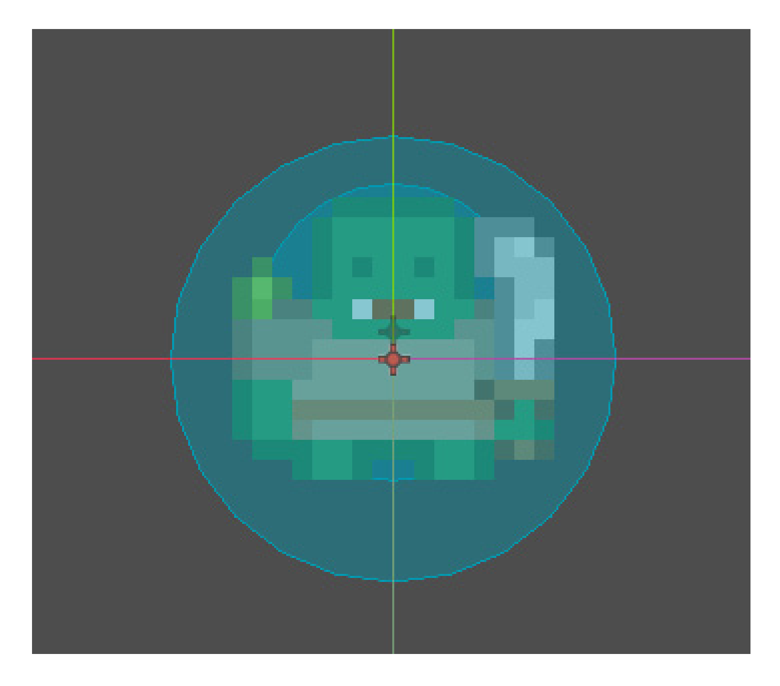


Рисунок 10.27 – Покрытие всего противника с помощью узла CollisionShape2D с некоторым запасом

1. Подключите сигнал **body_entered** к корневому узлу противника — Enemy.
2. Теперь используйте следующий фрагмент кода в качестве тела связанной функции в скрипте enemy.gd:

```
func _on_player_detection_area_body_entered(body: N
    if not body.is_in_group("player"):
        return
    body.get_hit()
    queue_free()
```

Код этой функции прост. Сначала мы проверяем, является ли тело, вошедшее в область, игроком. Мы можем сделать это просто с помощью следующей проверки:

```
body.is_in_group("player")
```

Таким образом, мы можем проверить, находится ли определённый узел в определённой группе. Если это тело не находится в группе **player**, мы возвращаемся из функции.

Но если тело является узлом игрока, то мы снимаем одно очко с его здоровья и освобождаем узел врага, который вступил с ним в контакт.

Отлично – теперь наш враг может нанести урон игроку, когда подойдет достаточно близко. Осталась только одна проблема: врагов столько, сколько мы смогли перетащить на сцену. Враги должны иметь возможность появляться автоматически и постоянно! В противном случае игра закончится очень быстро. Давайте сделаем автоматический спавнер, который будет создавать не только врагов, но и зелья здоровья.

Появление врагов и предметов коллекционирования

Автоматически создавать врагов или предметы коллекционирования на нашем игровом поле на самом деле сложнее, чем кажется на первый взгляд. Мы можем случайно выбрать место и создать что-то там. Однако, делая это, можно создать врага или предмет коллекционирования внутри стены или валуна. Хуже того, враг или предмет коллекционирования могут появиться за много километров от арены и области, где

они могут перемещаться, что сделает их бесполезными.

Мы могли бы решить эту задачу многими умными и абстрактными способами, но начнём мы с самого простого способа. Мы создадим наш собственный спавнер сущностей, который может создавать различные виды сущностей, врагов, коллекционных предметов или что-нибудь еще.

Создание структуры сцены

Элементарный способ решения проблемы расположения мест появления врагов — это задание определённых точек на арене, где враг может появиться не нарушая логику игры. Итак, вот что мы собираемся сделать в следующих шагах:

1. Создайте новую сцену, которая является производной от узла **Node2D** и назовите её **EntitySpawner**.
2. Сохраните эту сцену как **entity_spawner.tscn** в **parts/entity_spawner**.
3. К **EntitySpawner** добавьте ещё один узел **Node2D** и назовите его **Positions**. Позднее мы определим здесь все позиции, где мы будем что-то создавать:

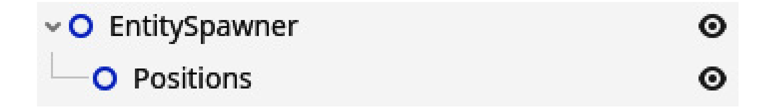


Рисунок 10.28 – Структура нашей сцены EntitySpawner

1. Перетащите экземпляр (инстанс) **EntitySpawner** на сцену **main.tscn** и переименуйте его в **EnemySpawner**.
2. Теперь щёлкните правой кнопкой мыши по **EnemySpawner** и выберите **Редактируемые потомки (Editable Children)**:

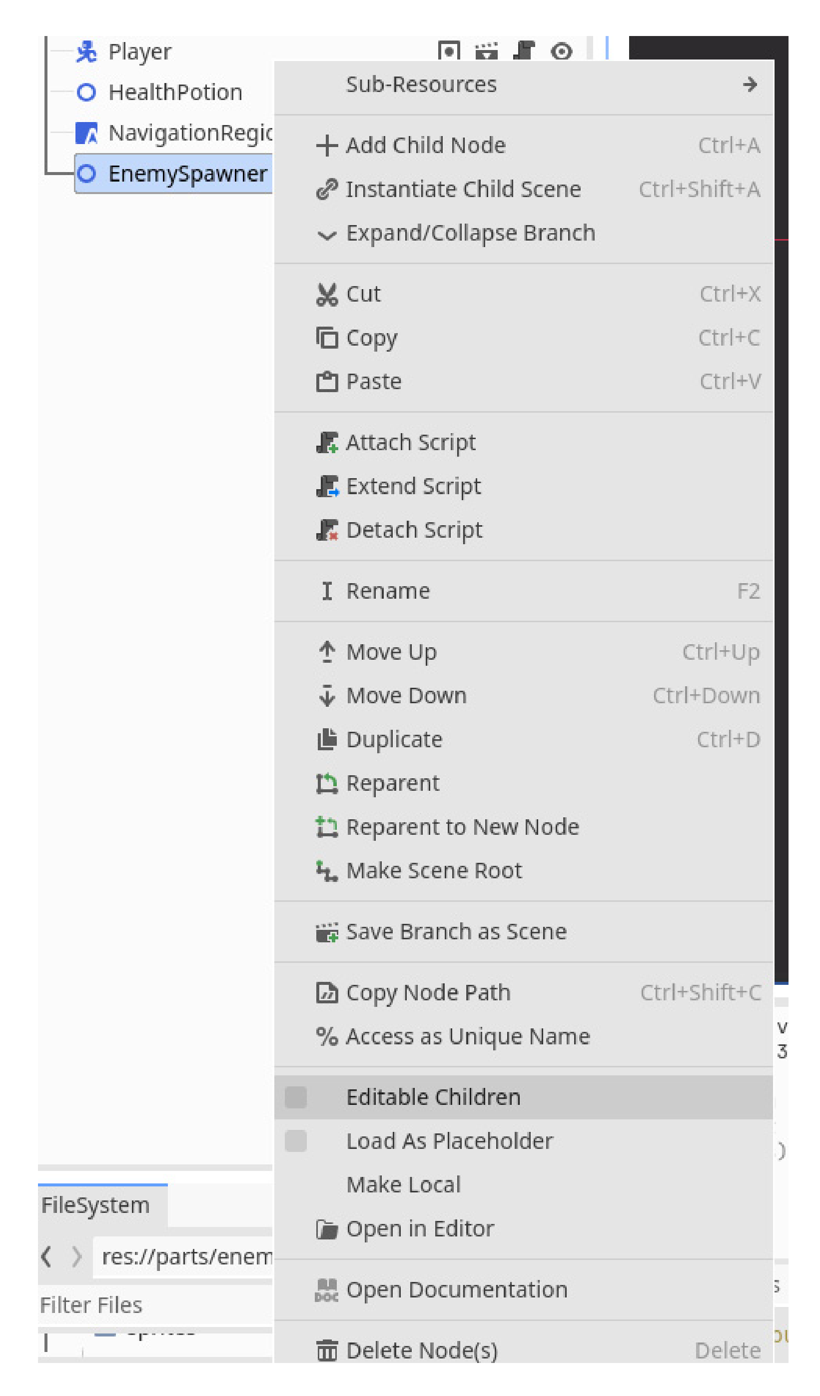


Рисунок 10.29 – Включение функции «Редактируемые потомки»

для непосредственного редактирования дочерних элементов экземпляра сцены

Вы увидите узел **Positions** , который является дочерним элементом сцены **EnemySpawner**:

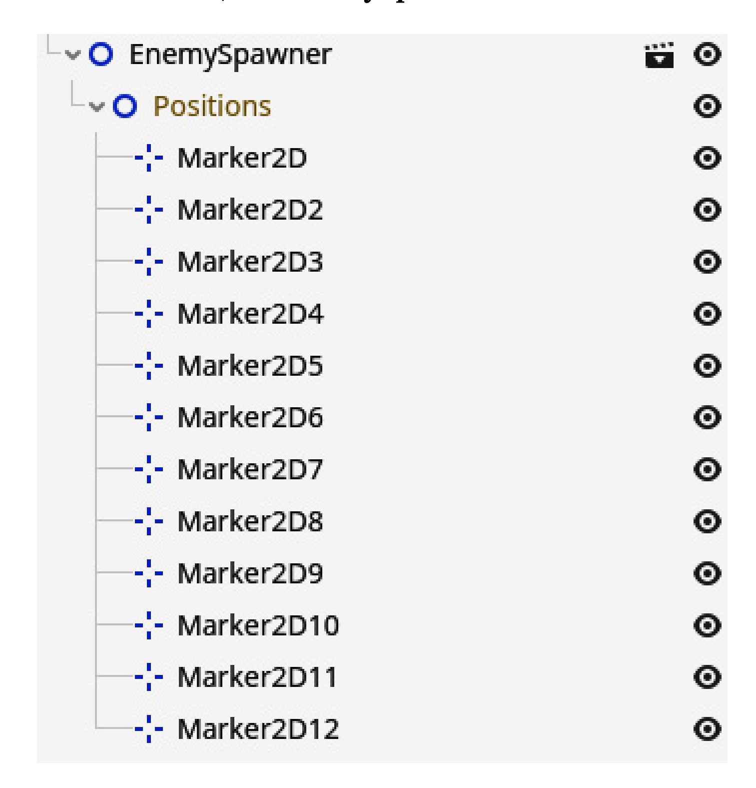


Рисунок 10.30 – Добавление узлов Marker2D, которые будут использоваться для позиций появления врагов

1. Теперь под этим узлом **Positions** добавьте несколько узлов **Marker2D** и разместите их в местах, подходящих для появления врагов:

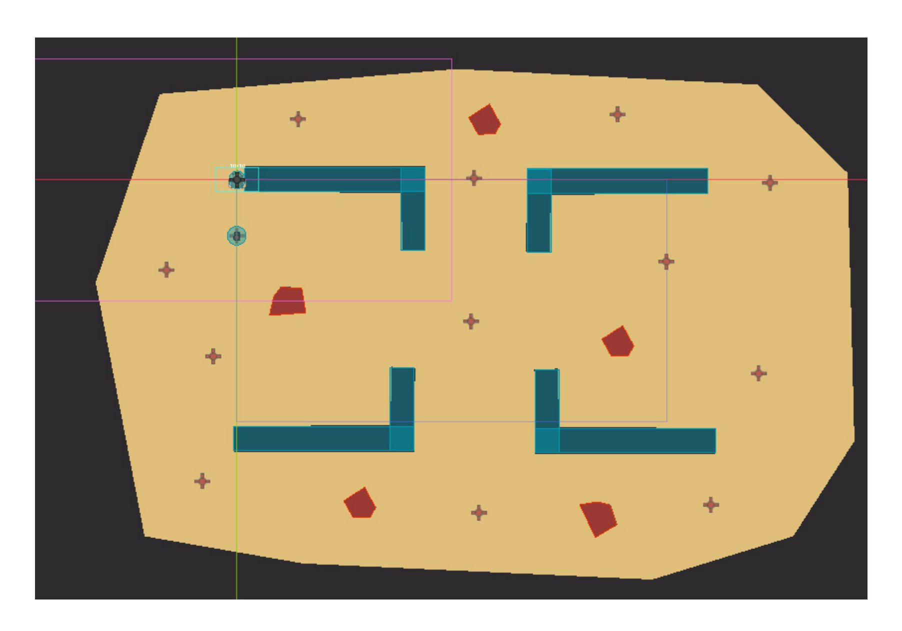


Рисунок 10.31 – Различные позиции, в которых я хотел, чтобы появлялись враги

До сих пор узел **EnemySpawner** было довольно просто настроить, но мы использовали некоторые новые вещи. Во-первых, мы включили **Редактируемые потомки (Editable Children)** на узле, который является полной сценой. Это открывает нам структуру всей сцены и упрощает редактирование отдельных узлов внутри. Это очень полезно для прямого использования сцен повторно.

Обратите внимание, что узлы под узлом **EnemySpawner** затенены. Это означает, что мы можем редактировать их, перемещать и т.д., как когда мы наследовали от коллекционной сцены, чтобы сделать зелье здоровья, но мы не можем удалить эти затененные узлы.

После включения редактирования потомков мы использовали новый тип узла: **Marker2D**. Это узел, который на самом деле ничего особенного не делает во время игры, но в редакторе он будет отображать маленький крестик, чтобы отметить место, в котором он расположен. Этот узел используется, если вам нужно отметить позицию, как мы делаем это здесь.

Написание базового кода

В коде мы сделаем всё довольно просто — предоставим функцию **spawn_entity()** которая создаёт новую сущность, будь то враг или зелье здоровья, в одной из определённых позиций:

```
extends Node2D
@export var entity_scene: PackedScene
@onready var _positions: Node2D = $Positions
func spawn_entity():
   var random_position: Marker2D = _positions.get_childr
   var new_entity: Node2D = entity_scene.instantiate()
   new_entity.position = random_position.position
   add_child(new_entity)
```

Однако, в этом коде есть несколько новых элементов. Первое, с чем мы сталкиваемся, — это экспортируемая переменная типа **PackedScene**. Эта переменная **PackedScene** по сути является определением любой сцены — файлом сцены. Любой файл сцены может заполнить эту переменную.

Разница между переменными PackedScene и Node

Переменная **PackedScene** представляет файл сцены, например файл **enemy.tscn**. Это шаблон, который мы можем использовать для создания новых узлов.

С другой стороны, переменная **Node** является строительным блоком дерева сцены и может быть экземпляром переменной **PackedScene**.

Переменную **PackedScene** можно рассматривать как класс, тогда как переменная **Node** — это экземпляр объекта этого класса.

Затем, позже, мы можем использовать эту упакованную сцену для создания новой сущности:

```
var new_entity: Node2D = entity_scene.instantiate()
```

Последнее, что нам нужно сделать, чтобы сделать эту новую инстанцированную сущность частью дерева сцены, — это добавить её к существующему узлу в дереве, поскольку, если мы не добавим её куда-нибудь в дерево сцены, она не будет использоваться в игре или при её выполнении.

Мы можем добавить новый узел в качестве дочернего к другому узлу, вызвав функцию **add_child()** на любом узле в дереве с этим новым узлом сущности в качестве параметра. Затем сущность будет добавлена в качестве дочернего к этому узлу. Здесь мы добавляем узел сущности в **EntitySpawner**:

```
add_child(new_entity)
```

Теперь сущность действительно помещена в дерево и, таким образом, в игру.

Чтобы выбрать случайную позицию, мы также делаем что-то новое. Сначала мы получаем массив потомков из узла **Positions** с помощью **get_children()**, который является массивом маркеров позиции. Затем, чтобы выбрать случайный элемент из этого массива, мы можем использовать функцию **pick_random()** для простого выбора одного случайного маркера позиции:

```
var random_position: Marker2D = _positions.get_children(
```

Это предоставит нам случайный узел **Marker2D** , который мы можем использовать для создания врага.

Чтобы заставить наш узел **EnemySpawner**, который находится в сцене **main.tscn**, порождать врагов, нам просто нужно перетащить сцену **enemy.tscn** поверх свойства **Entity Scene** на вкладке **Инспектор** узла **EnemySpawner**:

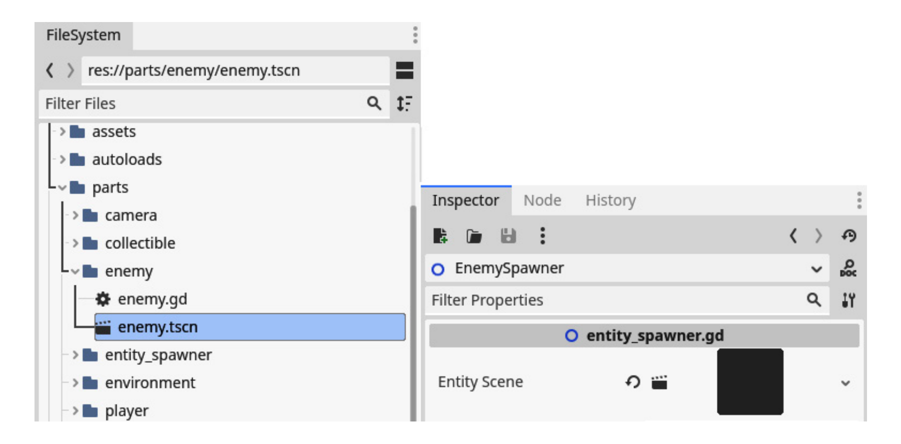


Рисунок 10.32 – Перетаскивание файла enemy.tscn в свойство Entity Scene

При такой настройке мы можем начать создавать сущности через фиксированные интервалы времени.

Автоматически создаваемые сущности

Теперь, когда у нас есть функция, которая может порождать сущность, нам всё ещё нужно запустить её в какой-то момент. Для этого мы воспользуемся узлом **Timer**. Этот узел отсчитывает определенное количество времени и выдаёт сигнал **timeout**, когда таймер заканчивается.

Давайте добавим узел **Timer** к сцене **EntitySpawner**:

1. Добавьте узел **Timer** в файл сцены **entity_spawner.tscn** и назовите его **SpawnTimer**.
2. Теперь подключите сигнал тайм-аута **timeout** к корневому узлу **EntitySpawner**.
3. В подключенной функции просто вызовите функцию **spawn_entity()**:

```
func _on_spawn_timer_timeout():
    spawn_entity()
```

4. Добавьте ссылку на узел **SpawnTimer** и переменную **export**, которая будет представлять интервал, с которым

мы будем создавать сущности в верхней части скрипта:

```
@onready var _spawn_timer: Timer = $SpawnTimer
@export var spawn_interval: float = 1.5
```

5. Теперь мы можем добавить две дополнительные функции, которые помогут нам запускать и останавливать таймер:

```
func start_timer():
   _spawn_timer.start(spawn_interval)
func stop_timer():
   _spawn_timer.stop()
```

6. Наконец, для автоматического запуска таймера в начале игры добавьте эту функцию **_ready()** в скрипт **EntitySpawner**:

```
func _ready():
   start_timer()
```

Важное примечание

Помните – когда мы говорим о сцене, мы говорим о целом файле сцены, например, файле **entity_spawner.tscn**. Когда мы говорим об узле, мы говорим о конкретном узле в файле сцены, например, узле **EntitySpawner**.

Функции запуска и остановки помогут, например, когда мы хотим остановить спавн врагов, когда игрок умирает. В основном они просто запускают и останавливают **_spawn_timer** напрямую. Вы можете видеть, что при запуске таймера мы можем указать время в секундах, которое будет использоваться как количество времени до истечения таймера.

Запустив игру сейчас, мы будем получать нового врага каждые 1,5 секунды. Отлично! Теперь, когда у нас есть поток врагов, давайте создадим несколько зелий, чтобы игрок мог исцелиться.

Создание зелий здоровья

Чтобы создавать коллекционные зелья здоровья, мы можем легко использовать тот же узел **EntitySpawner** который мы только что создали! Вот как это сделать:

1. Добавьте новый узел **EntitySpawner** к сцене **main.tscn** и назовите его **HealthPotionSpawner**.
2. Сделайте потомки этого спавнера редактируемыми и добавьте к узлу **Positions** узлы **Marker2D**, в которых вы хотите создавать зелья здоровья:

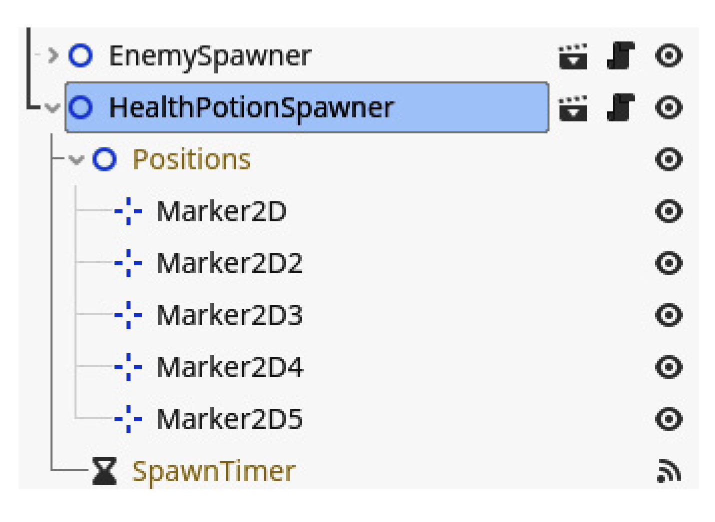


Рисунок 10.33 – Добавление узлов Marker2D, в позициях которых я хотел бы создавать зелья здоровья

1. Перетащите сцену **health_potion.tscn** в свойство **Entity Scene** спавнера.
2. Установите значение **Spawn Interval** спавнера на большее число, например 20, чтобы не создавать слишком много зелий здоровья:

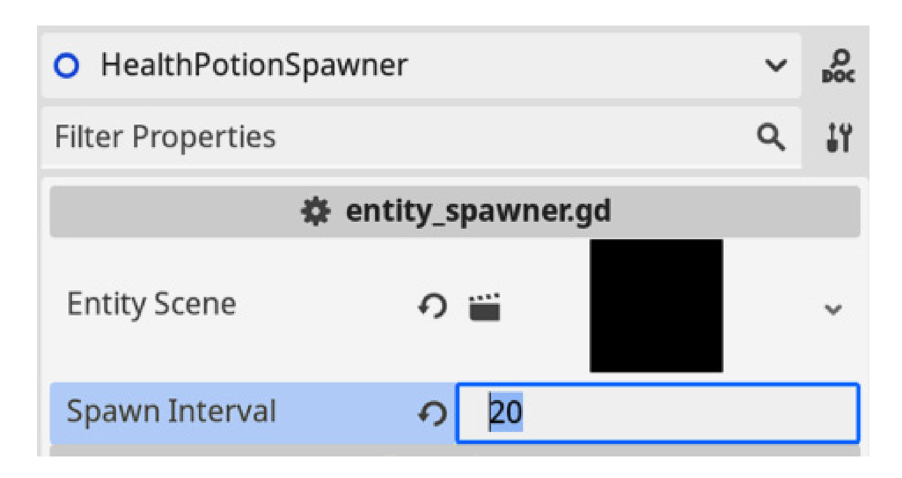


Рисунок 10.34 – Установка значения интервала появления зелий на 20 секунд

Вот и всё! Создавать новые вещи легко, если мы создадим сцену **EntitySpawner**, которую можно будет легко использовать повторно, не так ли?

Создание экрана конца игры (Game Over)

Теперь, когда враги могут наносить урон игроку, а здоровье игрока падает, нам нужно учесть сценарий, когда здоровье игрока достигает 0. Это будет означать конец игры. Мы добавим небольшой экран **Конец игры (Game Over)** который даст игроку возможность повторить попытку или вернуться в главное меню после смерти.

Создание базовой сцены

Как всегда, начнём с создания структуры сцены:

1. Создайте новую сцену с корневым узлом **CenterContainer**, назовите этот узел **GameOverMenu** сохраните сцену как **game_over_menu.tscn** в **parts/game_over_scene**.
2. Воссоздайте следующую структуру сцены:

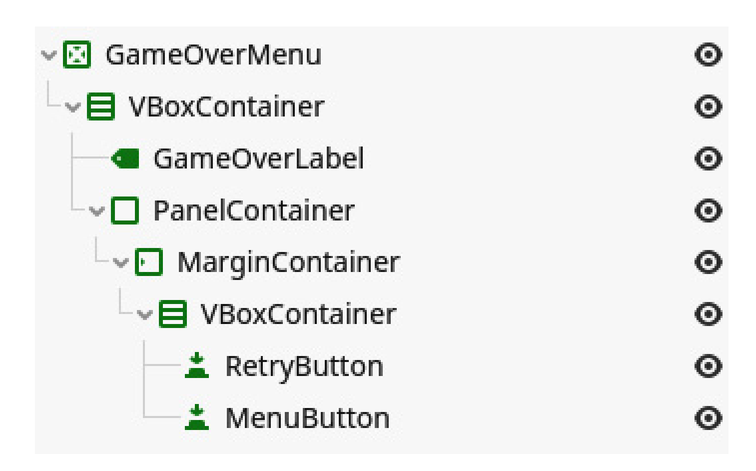


Рисунок 10.35 – Дерево сцены для меню Game Over

1. Заполните каждый элемент правильным текстом, увеличьте значение **Font Size** узла **GameOverLabel** и добавьте некоторое разделение к узлу **VBoxContainer**, который содержит две кнопки. Сделайте так, чтобы пользовательский интерфейс выглядел следующим образом:



Рисунок 10.36 – Как будет выглядеть меню Game Over

1. Теперь выберите корневой узел **GameOverMenu** и установите для него в типах **Anchor preset** значение **Full Rect**:

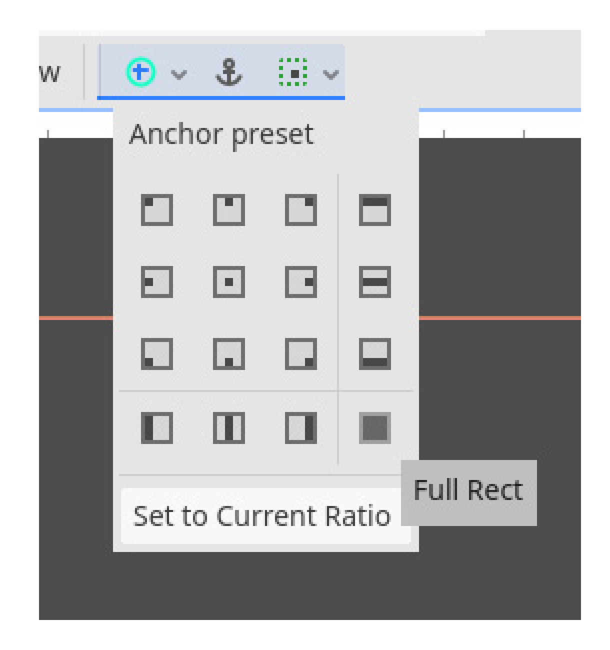


Рисунок 10.37 – Выбор Fill Rect из списка предустановок привязки

Теперь, когда у нас есть небольшое меню, давайте добавим его в сцену **main.tscn**:

1. В сцене **main.tscn** добавьте узел **CanvasLayer**.
2. Под этим узлом **CanvasLayer** добавьте наш недавно созданный узел **GameOverMenu**:

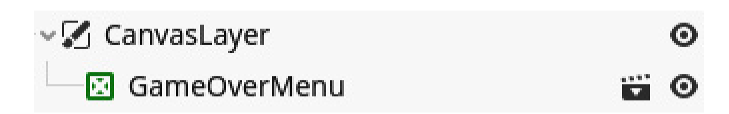


Рисунок 10.38 – Узел GameOverMenu добавлен в дерево сцены

1. Теперь скройте узел **GameOverMenu**, нажав на символ глаза рядом с именем узла. Мы хотим показывать это меню только тогда, когда игрок мертв:

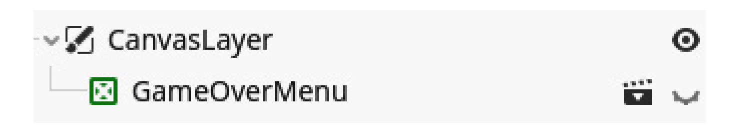


Рисунок 10.39 – Скрытие узла GameOverMenu путём нажатия на символ глаза рядом с его именем

Мы используем узел **CanvasLayer** для отображения нашего меню здесь, потому что этот узел гарантирует, что все его дочерние элементы отображаются поверх всего остального. Узел **CanvasLayer** не придерживается порядка отображения, который определяется порядком дерева сцены узлов. Внутри узла **CanvasLayer** его дочерние элементы снова придерживаются этого порядка. Это делает узел **CanvasLayer** очень подходящим для пользовательских интерфейсов в самой игре.

Это всё, что касается базовой структуры сцены; теперь нам нужно добавить немного логики в меню.

Добавление логики в меню Game Over

Скрипт для узла **GameOverMenu** очень прост. Всё, что мы хотим сделать, это добавить функциональность при нажатии кнопок. При нажатии кнопки **Play** мы перезагружаем основную игровую сцену, а при нажатии кнопки меню мы возвращаемся в главное меню.

Итак, подключите обе кнопки и загрузите нужную сцену в каждую из связанных с ними функций:

```
extends CenterContainer
func _on_retry_button_pressed() -> void:
   get_tree().reload_current_scene()
func _on_menu_button_pressed() -> void:
   get_tree().change_scene_to_file("res://screens/ui/mer
```

Важное примечание

Обратите внимание, что мы использовали новую функцию **reload_current_scene()** в дереве. Эта функция очень похожа на **change_scene_to_file()**, за исключением того, что она просто перейдет на ту же сцену, в которой мы находимся в данный момент, и нам не нужно загружать файл сцены, поскольку он,

очевидно, уже загружен.

Меню **Game Over** готово. Теперь нам просто нужно использовать его в игре.

Отображение меню Game Over при смерти игрока

Мы увидели, как можно подключиться к сигналам, которые испускают узлы. Но мы также можем создавать и испускать свои собственные сигналы! Мы воспользуемся этим, чтобы определить, когда игрок умирает:

1. В скрипте **player.gd**, прямо под строкой, содержащей ключевое слово **extends**, добавьте наш новый сигнал:

```
class_name Player extends CharacterBody2D
signal died
```

2. Давайте подадим этот сигнал и остановим движение игрока в сеттере здоровья — **health** когда **health** равно **0**:

```
    set(new_value):
        var new_health: int = clamp(new_value, 0, MAX
        if health > 0 and new_health == 0:
            died.emit()
            set_physics_process(false)
        health = new_health
        update_health_label()
```

Вы можете видеть, что для определения нового сигнала нам просто нужно использовать ключевое слово **signal**, за которым следует имя сигнала.

Затем, позже, мы можем просто испустить этот сигнал, вызвав функцию **emit()**для него. В некотором смысле, сигнал также является переменной.

Чтобы проверить, умер ли игрок, мы проверяем, больше ли текущее значение **health** чем 0, и равно ли 0 текущее значение

new_health. Таким образом, мы уверены, что сигнал **died** сработает только один раз, когда игрок переходит из живого в мертвое состояние. Мы не хотим, чтобы этот сигнал выдавался несколько раз, потому что это будет сигнализировать игре, что игрок умер больше одного раза, и создавать нежелательные побочные эффекты.

Затем мы также используем функцию **set_physics_process()** и передаём ей **false** в качестве единственного параметра. Это сообщает узлу, следует ли ему прекратить выполнение функции **_physics_process()**, и фактически остановит движение игрока, поскольку именно там находится весь наш код движения.

Теперь, когда узел **Player** выдает сигнал, когда погибает, мы можем подключиться к этому с помощью сцены **main.tscn**:

1. В сцене **main.tscn** выберите узел **Player**. Вы увидите, что появился новый сигнал — сигнал **died**, который мы определили в скрипте **player.gd**:

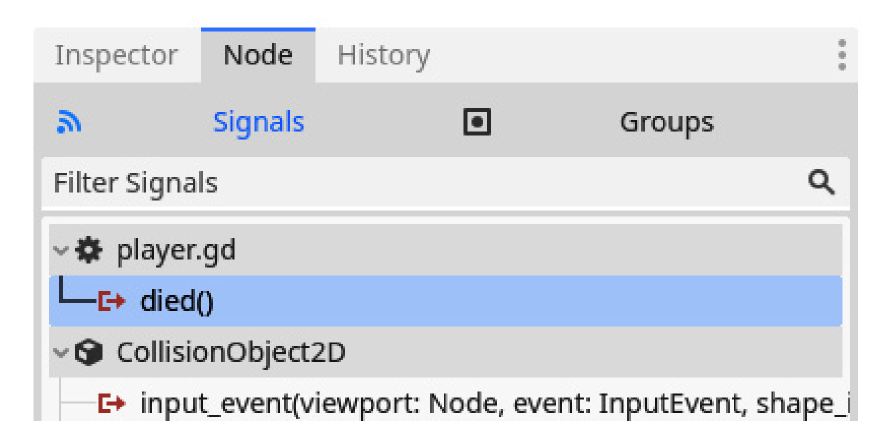


Рисунок 10.40 – Сигнал, который мы определили в скрипте player.gd, также появляется в меню сигналов

1. Добавьте пустой скрипт в узел **Main** сцены **main.tscn** и подключите к нему сигнал **died**.
2. В подключенной функции мы должны отобразить узел **GameOverMenu** и остановить узлы **EnemySpawner** и **HealthPotionSpawner**:

```
extends Node2D
@onready var _game_over_menu: CenterContainer = $Ca
@onready var _enemy_spawner: Node2D = $EnemySpawner
@onready var _health_potion_spawner: Node2D = $Heal
func _on_player_died() -> void:
   _game_over_menu.show()
   _enemy_spawner.stop()
   _health_potion_spawner.stop()
```

Этот скрипт довольно прост, поскольку ему нужно только управлять меню и останавливать некоторые спавнеры.

Мы рассмотрели много материала в этом разделе. Мы узнали, как можно использовать узлы **NavigationRegion2D** и **NavigationAgent2D**, чтобы заставить врагов двигаться к персонажу игрока. Мы использовали переменные **PackedScene** для создания сцен из кода. Мы использовали узел **Timer** для создания врагов и предметов коллекционирования по истечении определённого времени. Мы использовали узел **CanvasLayer** для отображения меню **Game Over** поверх игры. Мы создали собственный сигнал и подключились к нему. Мы знатно повеселились, и теперь пришло время игроку научиться защищаться!

Стрельба снарядами

Мы послали достаточно врагов на игрока, не дав ему возможности защитить себя. Давайте изменим это в этом разделе! Мы создадим снаряды, которые персонаж игрока автоматически выстрелит во врагов, чтобы убить их. Чтобы не усложнять задачу, мы заставим снаряд нацеливаться на цель, в которую мы пытаемся попасть; таким образом, он никогда не промахнётся.

Создание базовой сцены

Прежде чем мы сможем стрелять снарядами, нам нужно будет

построить базовую сцену, с которой мы будем работать. Давайте сделаем это прямо сейчас, выполнив следующие шаги:

1. Создайте новую сцену с узлом **Node2D** в качестве корневого узла и назовите его **Projectile**.
2. Создайте структуру сцены, как показано ниже:

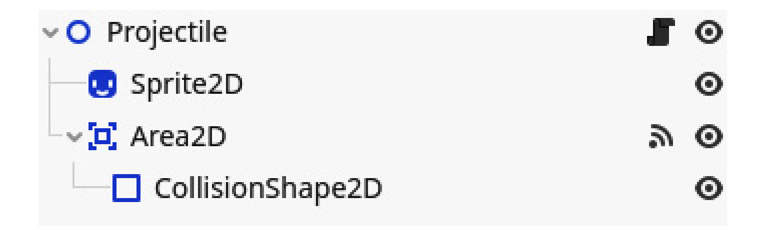


Рисунок 10.41 – Дерево сцены со снарядом

1. Используйте одну из текстур из **assets/sprites/projectils/** в качестве текстуры для спрайта. Не забудьте установить масштаб спрайта на **(3, 3)**:

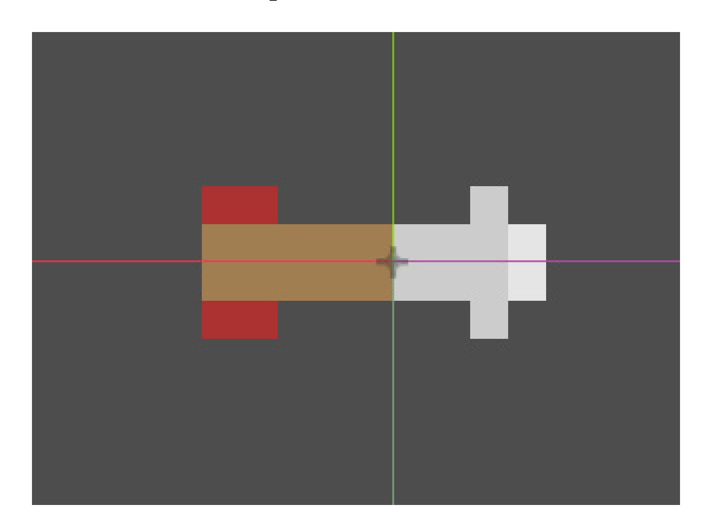


Рисунок 10.42 – Снаряд

1. Теперь используйте узел **CapsuleShape2D** для формы узла **CollisionShape2D** и убедитесь, что он покрывает спрайт:

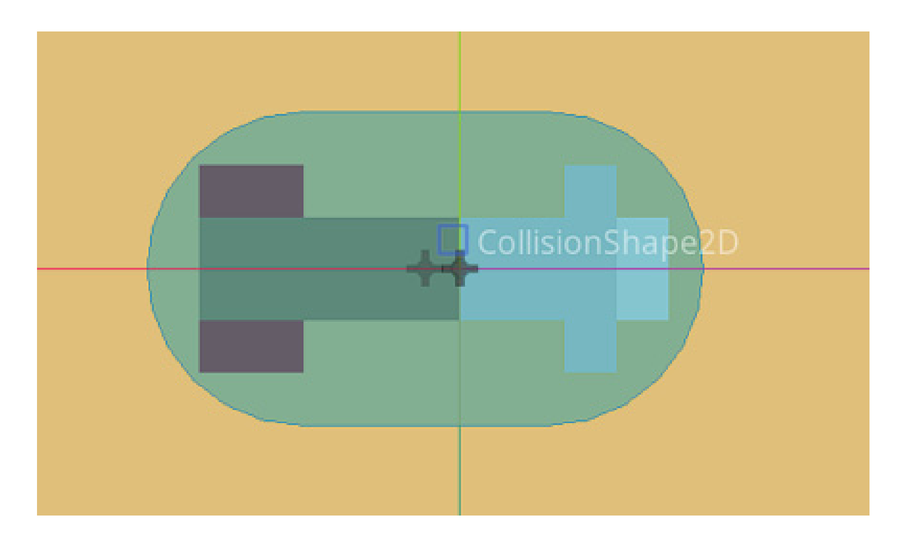


Рисунок 10.43 – Покрытие спрайта снаряда узлом CollisionShape2D

1. Мы будем использовать узел **Area2D** для определения того, попал ли снаряд во врага, поэтому переименуем этот узел области в **EnemyDetectionArea**.
2. Чтобы обнаружить узел **Enemy**, входящий в узел области **EnemyDetectionArea**, назовите третий 2D-слой **2D Physics** именем **Projectile**.
3. Установите свойство **Collision Mask** узла области **EnemyDetectionArea** для обнаружения слоя **Projectile**:

Collision
Layer
1 2 3 4 9 10 11 12 17 18 19 20
5 6 7 8 13 14 15 16 21 22 23 24
Mask
1 2 3 4 9 10 11 12 17 18 19 20
5 6 7 8 13 14 15 16 21 22 23 24

Рисунок 10.44 – Конфигурация слоя Collision для узла области EnemyDetectionArea

1. В сцене **enemy.tscn** установите свойство **Collision Layer** узла **Enemy** так, чтобы оно также находилось на слое **Projectile**:

Рисунок 10.45 – Конфигурация слоя столкновений для противника

Это всё, что нам нужно с точки зрения структуры сцены, поэтому давайте приступим к написанию поведения снаряда.

Написание логики снаряда

Далее следует код, который направляет снаряд к цели, уничтожает его при ударе и уведомляет врага о том, что он был поражен. Мы заставим снаряд всегда лететь прямо к своей цели; это упрощает нам задачу с точки зрения кода:

1. Присоедините скрипт с именем **projectile.gd** к корневому узлу **Projectile** и заполните его следующим кодом для его перемещения:

```
class_name Projectile
extends Node2D
@export var speed: float = 600.0
var target: Node2D
func _physics_process(delta: float):
```

```
    global_position = global_position.move_toward(ta
    look_at(target.global_position)
```

Мы уже видели большую часть этого кода, за исключением функции **look_at()**. Эта функция вращает узел, чтобы сориентироваться по отношению к точке в пространстве, которую мы ей предоставляем. Итак, здесь она вращает узел снаряда по отношению к положению цели.

2. Теперь подключим сигнал **body_entered** из узла **EnemyDetectionArea** к скрипту снаряда. Всё, что нам нужно сделать в подключенной функции, это уведомить врага о том, что он поражён снарядом, и уничтожить сам снаряд:

```
func _on_enemy_detection_area_body_entered(body: No
    body.get_hit()
    queue_free()
```

3. Наконец, в скрипт **enemy.gd** добавьте эту функцию **get_hit()**, которую мы хотим использовать, когда снаряд попадает во врага:

```
func get_hit():
    queue_free()
```

Это всё, что нам нужно с точки зрения кода на стороне самого снаряда.

Появление снарядов

Мы хотим, чтобы снаряд выстреливался автоматически время от времени. Чтобы добиться этого, нам нужно будет внести некоторые изменения в сцены игрока — **Player** и врага — **Enemy**:

1. Добавьте узел **Timer** к сцене **Player** и назовите этот новый узел **ShootTimer**.

2. Установите время этого узла **ShootTimer** на **0.5** и включите **Autostart**:

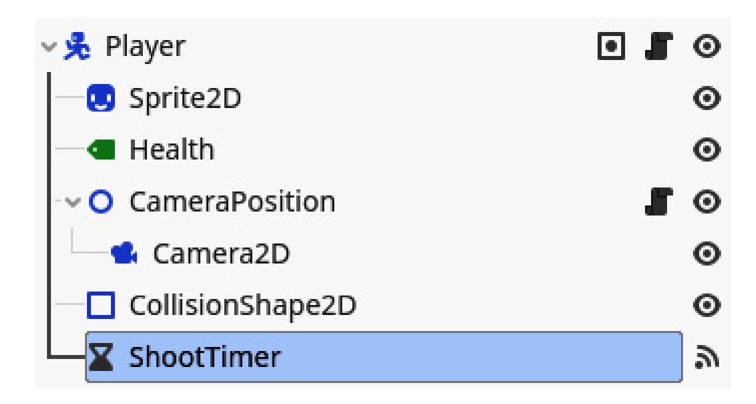


Рисунок 10.46 – Добавление узла таймера с именем ShootTimer к сцене игрока — Player

1. Далее в скрипте игрока предварительно загрузите сцену со снарядом вверху:

```
@export var projectile_scene: PackedScene = preload
```

2. Выбрав узел **Player**, перетащите файл **projectile.tscn** в свойство **Projectile Scene** на вкладке **Инспектор**.

Как и в случае с узлом **EntitySpawner**, мы экспортируем переменную типа **PackedScene**, которую мы можем заполнить из редактора и создать экземпляр позже, когда она нам понадобится. Однако на этот раз мы напрямую заполняем её сценой **projectile.tscn**. Функция **preload()** загружает эту сцену и помещает её в переменную **projectile_scene** , готовую к использованию. Но эта переменная также экспортируется, что означает, что если когда-нибудь мы захотим, чтобы игрок выстрелил другим типом снаряда, мы можем перетащить его сцену в поле этой переменной на вкладке **Инспектор** узла Player.

Теперь добавим логику, которая фактически создаёт снаряд:

1. В сцене **enemy.tscn** добавьте корневой узел в группу

enemy, как мы это делали ранее для сцены игрока — player. ТЭто позволит нам позже получить доступ ко всем узлам врагов:

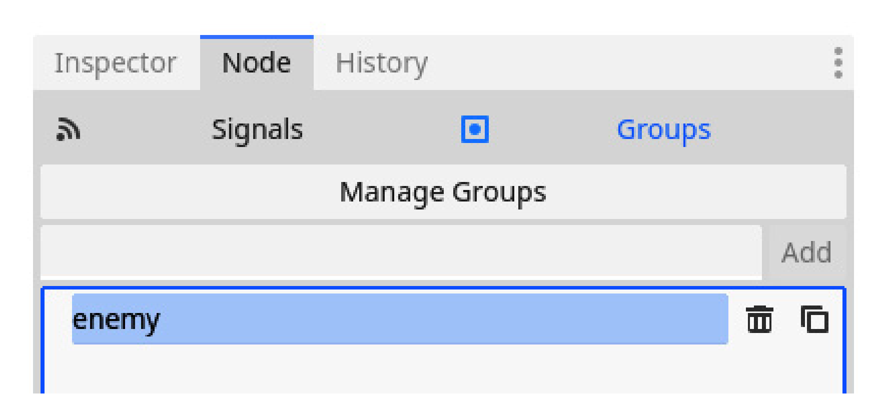


Рисунок 10.47 – Добавление узла enemy в группу enemy

1. Добавьте новую переменную **export** в начало скрипта **player.gd**. Эта переменная будет представлять дальность стрельбы игрока в пикселях:

```
@export var shoot_distance: float = 400.0
```

2. Теперь подключите сигнал тайм-аута (timeout) узла **ShootTimer** к скрипту узла **Player**.

   Код функции-обработчика этого подключенного сигнала:

```
func _on_shoot_timer_timeout():
    var closest_enemy: Enemy
    var smallest_distance: float = INF
    var all_enemies: Array = get_tree().get_nodes_in
    for enemy in all_enemies:
        var distance_to_enemy: float = global_positio
        if distance_to_enemy < smallest_distance:
            closest_enemy = enemy
            smallest_distance = distance_to_enemy
    if not closest_enemy:
        return
```

```
    if smallest_distance > shoot_distance:
        return
    var new_projectile: Projectile = ProjectileScene
    new_projectile.target = closest_enemy
    get_parent().add_child(new_projectile)
    new_projectile.global_position = global_position
```

3. Мы также должны остановить узел **ShootTimer**, когда игрок умирает. Для этого мы кэшируем узел **ShootTimer** в верхней части скрипта игрока и остановим его, когда здоровье игрока достигает 0::

```
@onready var _shoot_timer = $ShootTimer
@export_range(0, MAX_HEALTH) var health: int = 10:
    set(new_value):
        # Code to update the health
        if health > 0 and new_health == 0:
            # Code when player dies
            shoot_timer.stop()
```

Смысл этой функции заключается в том, что мы сначала получаем список всех врагов, используя функционал групп. Затем мы проходим по каждому из врагов в списке, чтобы увидеть, насколько далеко они находятся от игрока. При выполнении этого цикла мы всегда сохраняем ближайшего врага вместе с его дистанцией до игрока. Таким образом, мы знаем, что в итоге получим врага, который ближе всего к персонажу игрока.

Результатом этого алгоритма может стать отсутствие выбранного врага. Вот почему нам нужно убедиться, что **closest_enemy** не является случайно пустым, и нужно вернуться из функции, если это всё же так.

После всего этого мы создаём новый снаряд, задаём ему цель, добавляем его в дерево сцены и привязываем его позицию к позиции игрока.

Вот и всё, что касается создания снарядов! Теперь вы можете запустить игру и попытаться выжить как можно дольше. Мы

также увидели более сложный код с алгоритмом поиска ближайшего узла из любого другого узла и способ предварительной загрузки сцены (preload) в скрипте.

Сохранение рекордов в автозагрузках

Теперь, когда игрок может дать отпор и выжить, хорошо бы нам дать игроку цель для достижения — что-то, что заставит его играть снова и снова. Мы могли бы добавить рекорд — например, количество времени, которое игрок смог выжить. Затем игрок может попытаться улучшить своё время или сравнить время со временем своих друзей.

Для этого мы будем использовать автозагрузку. Это узел, который инициализируется в начале игры и будет существовать на протяжении всего выполнения игры.

Использование автозагрузки

Время выживания (survival time) должно храниться так, чтобы к нему был легкий доступ из любой точки игры. Мы должны иметь возможность изменить его после смерти игрока, а также отобразить счет в главном меню, например.

Обычные узлы и сцены должны управляться нами, программистами. Но есть другой тип узлов, которые мы могли бы использовать: автозагрузки (autoloads). Автозагрузка — это сцена или скрипт, которые всегда загружены в игру. Движок Godot инициирует эту сцену для нас каждый раз, когда мы запускаем игру.

Узел или скрипт, который загружается автоматически, будет существовать, пока игра запущена. Ранее, при использовании **get_tree().change_scene_to_file()** для смены сцен, всё содержимое текущей сцены удалялось из дерева сцены и переключалось на новую сцену. Однако автозагрузки не разделяют ту же участь; они остаются на месте и сохраняют

значения всех своих переменных.

Важное примечание

Хотя автозагрузки — это здорово, их не следует использовать неправильно или злоупотреблять ими. Их следует использовать только для систем, которые действительно являются глобальными, например, автозагрузка **HighscoreManager**, которую мы собираемся создать в этом разделе.

Мы пока не будем сохранять автозагрузку рекордов в файле; мы сделаем это в *Главе 15*. Сейчас мы просто хотим сохранять и загружать автозагрузку рекордов во время работы игры.

Создание автозагрузки достижений — HighscoreManager

Чтобы создать автозагрузку, нам сначала нужно создать обычную сцену или скрипт. Поскольку нам на самом деле не нужна целая сцена для отслеживания рекорда, который по сути является просто числом, мы напишем скрипт. Когда движок Godot инициирует нашу игру, он создаст узел и прикрепит к нему наш скрипт. Следующие шаги иллюстрируют процесс создания автозагрузки:

1. Создайте новую папку **autoloads/** в корне проекта.
2. Добавьте в эту папку новый скрипт под названием **highscore_manager.gd**.

   Скрипт **HighscoreManager** будет довольно простым и понятным:

   ```
   extends Node
   var highscore: int = 0
   func set_new_highscore(value: int):
      if value > highscore:
         highscore = value
   ```

Этот код определил переменную **highscore** и функцию

set_new_highscore(). Эта функция проверяет, больше ли новый счёт, чем текущий. Если да, мы сохраняем этот новый, более высокий счёт; в противном случае нам не нужно беспокоиться.

Теперь давайте установим этот скрипт как автозагрузку:

1. Откройте настройки проекта и перейдите на вкладку **Автозагрузка (Autoload)**.
2. Нажмите кнопку со значком папки, чтобы выполнить поиск файла:

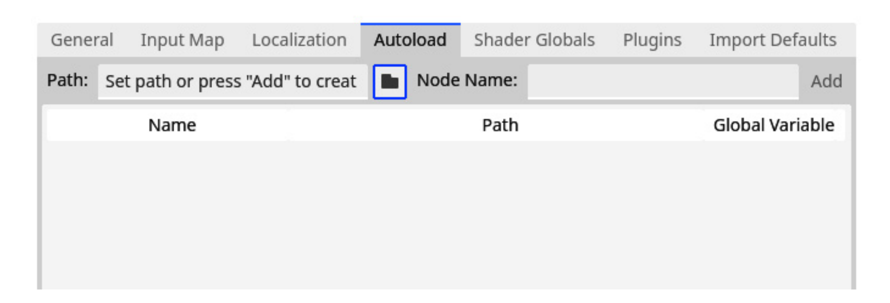


Рисунок 10.48 – Нажатие на значок папки для выбора файла, который вы хотите добавить в качестве автозагрузки

1. Перейдите к скрипту **autoloads/highscore_manager.gd**.
2. Выберите его и нажмите **Открыть (Open)**.
3. Теперь вернитесь на панель **Автозагрузка** в настройках проекта и нажмите кнопку **Добавить (Add)**.

Вот и все по настройке нашей автозагрузки. Вы увидите, что автозагрузка **Highscore** теперь отображается в списке автозагрузок:

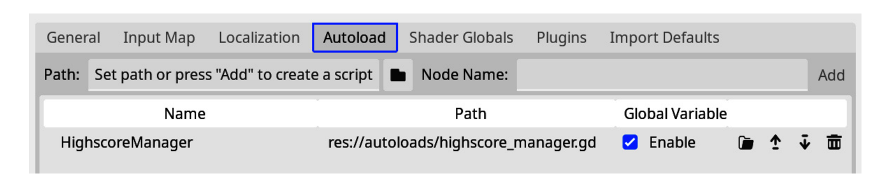


Рисунок 10.49 – Скрипт highscore_manager.gd загружается как автозагрузка

Помимо просмотра скрипта в списке автозагрузок, есть ещё

один способ проверить, есть ли там автозагрузка.

Использование сцены как автозагрузки

Как скрипты, так и полные сцены могут быть автозагружены. Чтобы использовать сцену, загрузите её так же, как мы только что сделали для скрипта.

Автозагрузки в удалённом дереве

Как было сказано ранее, автозагрузки создаются движком Godot при запуске игры. Поэтому мы не можем видеть их в отдельных сценах, но они должны быть в удалённом дереве при запуске игры.

Запустите игру с помощью кнопки **Запустить проект (Run Project)** или любую сцену игры с помощью кнопки **Запустить текущую сцену (Run Current Scene)**. Откройте удалённое дерево, нажав кнопку **Удалённый (Remote)**, и вы увидите узел **HighscoreManager**. Это наша автозагрузка **HighscoreManager**!

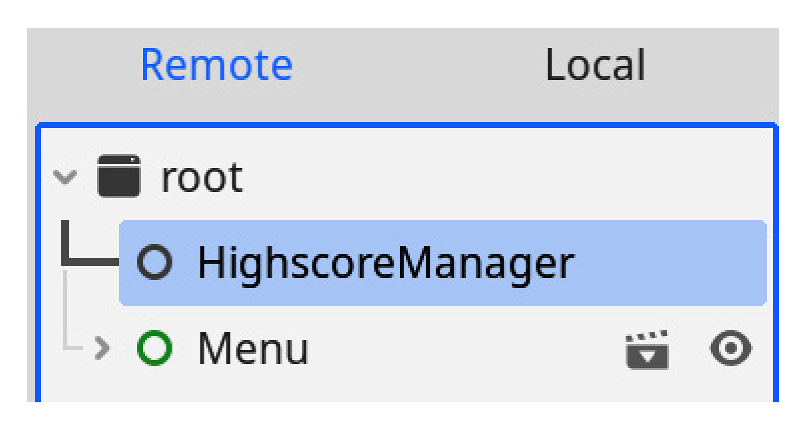


Рисунок 10.50 – Мы видим узел HighscoreManager в удалённом дереве

Теперь, когда мы настроили автозагрузку **HighscoreManager**, давайте используем её в игре и сохраним несколько рекордов!

Добавление пользовательского

интерфейса в главное меню и игровую сцену

Во-первых, нам нужно убедиться, что игрок знает, каков его счёт во время игры. Поскольку мы сказали, что счёт будет количеством времени, которое игрок сможет выжить, мы покажем этот счёт, добавив таймер на экран:

1. В сцене **main.tscn**под существующим узлом **CanvasLayer** добавьте узел **CenterContainer** и назовите его **TimerUI**.
2. Для узла **TimerUI** выберите якорь **Сверху по всей ширине (Top Wide)**, чтобы он оставался в верхней части экрана:

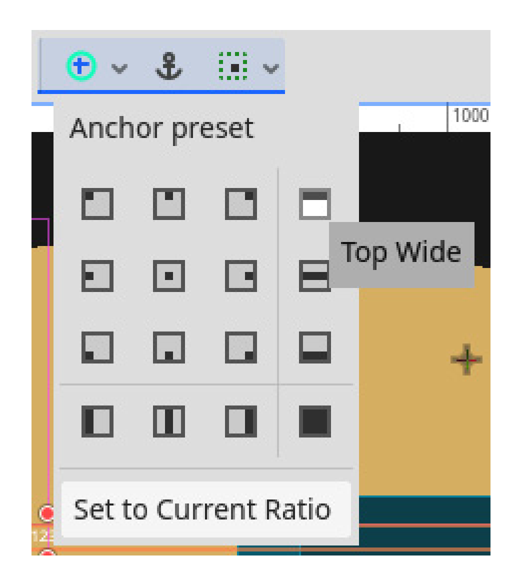


Рисунок 10.51 – Выбор Сверху по всей ширине (Top Wide) из списка Пресет якорей

1. Добавьте узел метки **Label** в **TimerUI** и назовите его **TimeLabel**:

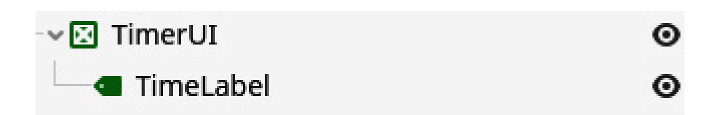


Рисунок 10.52 – Структура сцены для показа нашего таймера

1. Давайте заполним эту метку фиктивным временем "**123**" , чтобы увидеть, как будет выглядеть счёт, когда она будет заполнена.
2. Измените размер шрифта (Font Size) этой метки на больший, например, **30** пикселей.

Теперь нам нужно зарегистрировать таймер в игровом скрипте **main.gd**:

1. Сначала кэшируем ссылку на узел **TimeLabel** в верхней части скрипта и добавляем переменную, в которой будем хранить текущее прошедшее время:

```
@onready var _time_label: Label = $ CanvasLayer/Tim
var _time: float = 0.0:
   set(value):
      _time = value
      _time_label.text = str(floor(_time))
```

2. Теперь нам нужно только обновить значение переменной **_time**. Мы сделаем это в функции **_process()**, добавив дельту к текущему времени:

```
func _process(delta: float):
   _time += delta
```

3. Наконец, нам нужно будет отправлять это время всякий раз, когда игрок умирает, и прекращать подсчёт времени. Поэтому измените функцию, которая связана с сигналом **died** от игрока, чвключив в неё следующие две строки:

```
func _on_player_died() -> void:
   _game_over_menu.show()
   _enemy_spawner.stop()
   _health_potion_spawner.stop()
   set_process(false)
```

```
    HighscoreManager.set_new_highscore(_time)
```

Вот и всё, что касается ссылки на рекорд в самой игре. Теперь мы займёмся отображением рекорда в главном меню.

Использование рекорда в главном меню

Теперь, когда мы можем создавать новые рекорды, давайте отобразим наивысший результат в меню:

1. Откройте сцену **menu.tscn**.
2. Добавьте новый узел **Label** к узлу **VBoxContainer**, содержащему кнопки **Play** и **Exit** , и назовите его **HighscoreLabel**.
3. Теперь в скрипт **menu.gd** добавьте следующий код:

```
@onready var highscore_label: Label = $CenterContai
func _ready():
   highscore_label.text = "Рекорд: " + str(Highscor
```

В результате в меню теперь будет отображаться текущий рекорд:



Рисунок 10.53 – Главное меню с добавленной меткой HighscoreLabel

Ничто из этого кода не является для нас новым. Сначала мы сохраняем узел **HighscoreLabel** в переменной **highscore_label**. Затем, когда сцена меню готова, мы заполняем **HighscoreLabel** строкой, содержащей текущий наивысший счёт.

Это было наше знакомство с автозагрузками. Мы увидели, как легко добавить скрипт или сцену в качестве узла, который всегда загружается в начале нашей игры, без необходимости управлять этим узлом самостоятельно. Затем мы использовали эту автозагрузку через её глобальную переменную для сохранения информации между различными сценами.

Дополнительные упражнения – Заточка топора

1. Враги появляются с медленной, фиксированной скоростью. Это может немного надоесть, потому что сложность никогда не увеличивается. Сделайте так, чтобы враги появлялись всё быстрее и быстрее после каждого раунда. Простой способ сделать это — выполнить следующие шаги:

    1. Добавьте **start_interval**, **end_interval** и **time_delta** в качестве экспортированных переменных в узел **EntitySpawner**. Переменная **start_interval** будет временем, которое мы используем между порождением сущностей в начале игры, **end_interval** будет конечным значением, а **time_delta** — это приращение, с которым мы перейдем от переменной **start_interval** к переменной **end_interval**:

Рисунок 10.54 – Новые экспортированные переменные для узла EntitySpawner

1. Теперь отслеживайте в отдельной переменной **_current_spawn_interval**, время появления следующего врага. Установите **_current_spawn_interval** равным переменной **start_interval** начале игры. Эта переменная заменяет старую переменную **spawn_interval**.
2. Каждый раз, когда мы создаем сущность в функции **spawn_entity**, добавляем переменную **time_delta** к переменной **_current_spawn_interval**. Однако убедитесь, что вы не вышли за пределы **end_interval**.
3. Затем, всё ещё находясь в функции **spawn_entity()**, снова запустите **_spawn_timer**, но с новой переменной **_current_spawn_interval**, вызвав **start_timer()** ещё раз. Для узла **HealthPotionSpawner** вам придётся установить **time_delta** на **0.0**.
4. Меню, которое появляется, когда игрок умирает, довольно неинформативно. Добавьте красивую метку (eptk Label), чтобы показать счёт, который только что набрал игрок.

Итоги

Мы узнали и создали много разных вещей в этой главе. Во-первых, мы узнали всё об узлах управления — **Control** и о том, как использовать их для создания главного меню для нашей

игры. Затем мы создали несколько испытаний в игре с врагами, которые пытаются остановить игрока. Мы даже заставили этих врагов ловко перемещаться по игровому полю с помощью свойства **NavigationServer**. Чтобы дать игроку возможность защитить себя, мы создали снаряды, которые автоматически выстреливаются по таймеру. Наконец, мы добавили небольшую систему рекордов, которая сохраняет текущий рекорд в автозагрузке, чтобы у игрока был стимул переиграть игру и попытаться побить свое лучшее время.

В следующей главе мы сделаем кое-что очень интересное: сделаем нашу игру многопользовательской!

Опрос

- Узлы управления **Control** используются для создания пользовательских интерфейсов, таких как меню. Для каждого из следующих сценариев укажите узел управления **Control**, который мог бы выполнить эту работу:
- Показ длинного фрагмента текста
- Группировка других узлов **Control** в центре экрана
- Показ кнопки начала игры
- Какой узел мы добавили к сцене **Enemy**, чтобы враг нашёл путь к игроку?
- Допустим, у нас есть фрагмент кода, в котором мы определяем сигнал, называемый **shot**, указывающий на то, что мы выстрелили снарядом:

  ```
  signal shot
  ```

  Напишите строку кода, необходимую для подачи этого сигнала.

- Как загрузить сцену в переменную, используя код?
- Как сделать скрипт доступным глобально?

11

Совместная игра в многопользовательском режиме

Играть в игры самостоятельно очень весело. Я провёл много часов, исследуя экзотические миры, приобретая новые навыки и переживая глубокие сюжетные линии самостоятельно. Но где игры действительно блистают, по сравнению с другими формами медиа, так это в возможности игрока создавать свои собственные истории. Ничто не позволяет игрокам создавать свою собственную историю так, как возможность играть с другим реальным человеком. От сотрудничества и напряженных моментов, когда вы пытаетесь помочь друг другу в таких играх, как World of Warcraft или Rocket League, до соперничества и запугивания друг друга в таких играх, как Call of Duty или Gran Turismo. Человеческое поведение по-прежнему вызывает больше эмоций, чем взаимодействие с полностью вымышленным миром.

В этой главе мы рассмотрим следующие основные темы:

- Ускоренный курс по компьютерным сетям
- Использование **MultiplayerSynchronizer** и **MultiplayerSpawner**
- Запуск игры на нескольких компьютерах

В этой главе мы реализуем сетевой многопользовательский режим. Это означает, что два человека смогут играть вместе через Интернет. Теперь, из-за того, как работают сети, и мы всё ещё хотим быть в безопасности, мы сможем играть только через **локальную сеть — local area network** (**LAN**). Это означает, что люди, подключённые к одной сети Wi-Fi, например, смогут играть вместе.

Причина, по которой вам не стоит запускать глобально доступный сервер с вашего персонального компьютера, довольно проста: вы не хотите риска взлома вашего компьютера. Хотя есть способы сделать это безопасным способом, это выходит за рамки этой книги.

Технические требования

Как и прежде, готовый код можно найти в репозитории GitHub в подпапке для этой главы: https://github.com/PacktPublishing/Learning-GDScript-by-Developing-a-Game-with-Godot-4/tree/main/chapter11.

Ускоренный курс по компьютерным сетям

В этом разделе я хотел бы дать вам краткий экспресс-курс по компьютерным сетям. Поскольку Godot делает многое из коробки, нам не нужно быть полными сетевыми волшебниками, чтобы реализовать простые многопользовательские игры. Это значит, что вы можете пропустить этот раздел и сразу начать с фактической реализации многопользовательских узлов и кода. Однако я рекомендую продолжить чтение, если вам интересно высокоуровневое объяснение того, почему мы делаем всё именно так.

Компьютеры в сетях общаются друг с другом через многоуровневую модель. На верхнем уровне находится конечное приложение — игра. Нашей игре необходимо отправлять информацию из одного экземпляра игры, запущенного на одном компьютере, в другой экземпляр игры, запущенный на другом компьютере, также называемом **другой машиной (another machine)**. Этот верхний уровень называется **прикладным уровнем (Application Layer)**. Между этими компьютерами может находиться обширная сеть взаимосвязанных серверов, маршрутизаторов и другой сетевой инфраструктуры. Эта сеть является самым нижним уровнем,

называемым **физическим уровнем (Physical Layer)**.

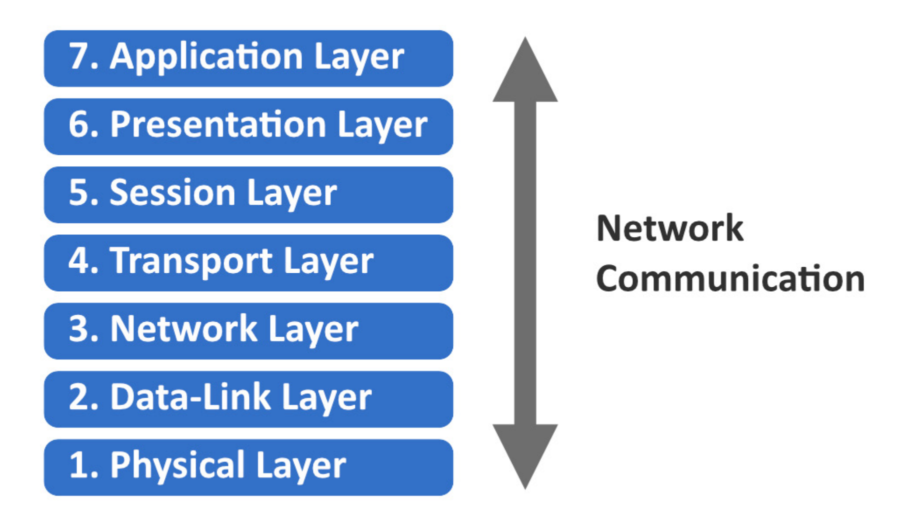


Рисунок 11.1 – Семь уровней компьютерной сети

Между прикладным уровнем и физическим уровнем есть несколько других уровней. Эти уровни обеспечивают отправку и получение данных между всеми звеньями в цепочке, которые необходимо предпринять для передачи этого пакета данных с компьютера A на компьютер B, и каждый из них служит своей цели.

Хотя Godot предоставляет нам большую гибкость, не каждый уровень одинаково важен для нас в данный момент. Давайте подробнее рассмотрим два сетевых уровня: транспортный и прикладной.

Что такое транспортный уровень?

Первый слой, который мы рассмотрим, — это **транспортный слой (Transport Layer)**, четвертый слой компьютерных сетей. Этот слой, помимо прочего, отвечает за принятие решения о том, как разбить данные, которые мы хотим отправить, на более мелкие пакеты данных, гарантируя, что пакеты данных будут переданы с одного конца на другой, неповреждёнными.

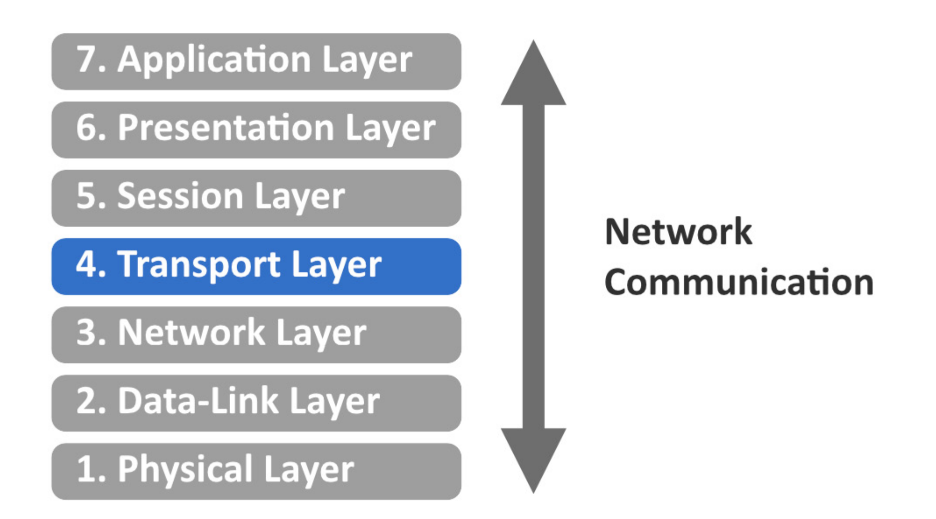


Рисунок 11.2 – Транспортный уровень — четвёртый уровень в компьютерных сетях.

Для выполнения этих обязанностей были изобретены различные протоколы, которые способны выполнять их с разной степенью надёжности. **Протокол (protocol)** — это, по сути, набор правил, с помощью которых компьютеры могут общаться друг с другом.

Например, если мы отправляем пакет данных с компьютера A на компьютер B, мы могли бы просто отправить его и надеяться на лучшее. Однако наш пакет данных может случайно потеряться где-то в огромном интернете. Сервер забудет отправить его с одного соединения на другое, кабель отсоединится или может произойти любая другая ошибка.

Как же нам убедиться, что данные, которые мы отправляем, действительно приходят? Ну, мы могли бы запросить подтверждение от принимающего компьютера. Но что, если это подтверждение где-то потеряется? Ну, мы могли бы сделать двойное подтверждение, по одному от каждого участника коммуникации.

Все эти правила просто решают проблему обеспечения отправки и получения пакета данных, но есть и много других проблем, которые нам нужно преодолеть. Как вы видите, эти

протоколы быстро становятся сложными. К счастью, умные люди уже подумали обо всём этом за нас.

В играх используются два основных протокола:

- **Протокол управления передачей (Transmission Control Protocol)** (**TCP**): TCP — это протокол транспортного уровня , который гарантирует, что каждый отправленный пакет будет получен. Но для этого протоколу требуется больше времени, отправляя подтверждения туда и обратно.
- **Протокол пользовательских датаграмм (User Datagram Protocol)** (**UDP**): UDP — это протокол транспортного уровня, которому всё равно , приходят пакеты или нет. Он просто отправляет их по соединению в надежде, что они придут, что большинство из них и должны делать. Это намного быстрее, чем TCP, но менее надежно.

Godot Engine может работать как с использованием TCP, так и UDP и даже может переключаться между ними для разных типов данных, в зависимости от того, насколько важна гарантированная доставка. Для нашей игры мы будем использовать как UDP, так и TCP для разных типов данных.

Что такое прикладной уровень?

Уровень приложений (Application Layer) — это самый высокий уровень в сетевых уровнях. Это когда мы фактически используем данные, которые получили в игре. Кроме того, здесь у нас есть выбор: даже если у нас есть данные, как мы собираемся организовать компьютеры, к которым мы подключены?

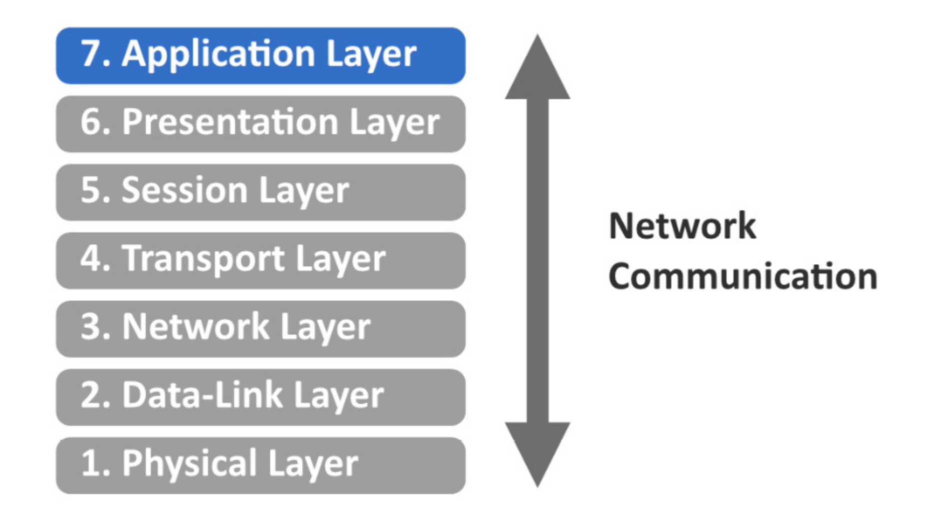


Рисунок 11.3 – Уровень приложений – седьмой уровень в компьютерных сетях.

Для игр преобладают две сетевые архитектуры: одноранговая (peer-to-peer) и клиент-серверная (client-server).

Одноранговая сеть (peer-to-peer)

В одноранговой сети каждый компьютер может общаться с любым другим компьютером и задавать ему вопросы. Они все равны и равноправны. Например, компьютер А может попросить компьютер B сообщить, в каком месте находится его игровой персонаж. Затем компьютер B отправит эти данные, чтобы компьютер А мог показать своему пользователю, где в игровом мире находится игровой персонаж компьютера B.

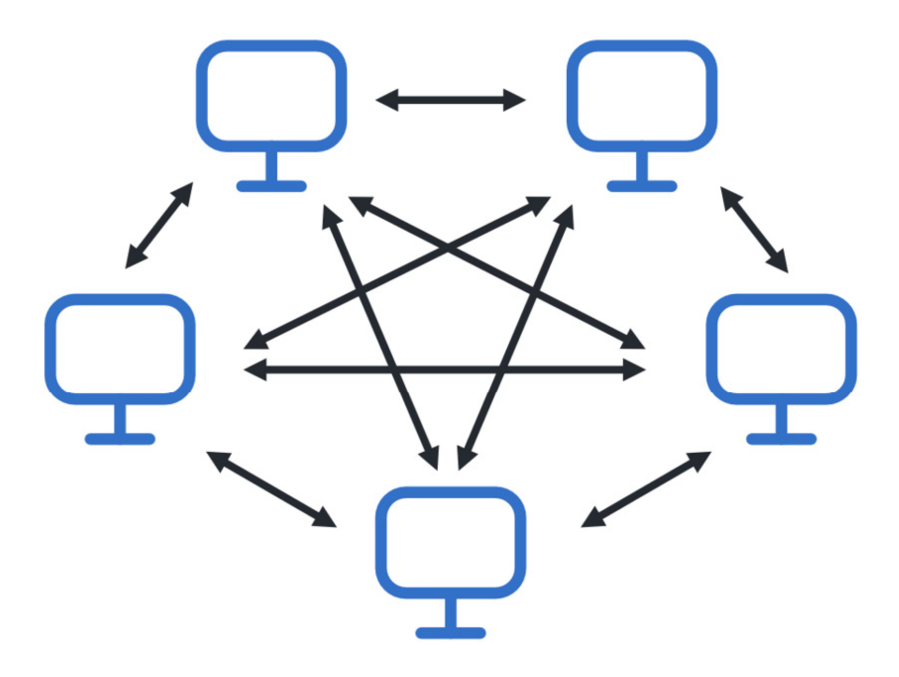


Рисунок 11.4 – В одноранговой сети каждый компьютер может взаимодействовать с другими компьютерами

Это решение довольно элегантно, поскольку все компьютеры равны и имеют одинаковое количество прав. Однако нам также нужно быть бдительными – что, если компьютер B используется хакером и лжёт другим компьютерам? Вместо того, чтобы сообщать о местоположении игрока в соответствии с правилами игры, компьютер B указывает позиции, до которых невозможно добраться; возможно, они телепортируют своего персонажа игрока. Это довольно серьёзная проблема. Следующая сетевая архитектура пытается решить эту проблему.

Клиент-серверная сеть

Вместо того, чтобы относиться ко всем компьютерам как к равным, мы могли бы сделать один из компьютеров центром для всех коммуникаций. Каждый раз, когда любой из компьютеров в сети захочет получить информацию, например, местоположение персонажа другого компьютера, ему придется спрашивать этот центральный компьютер. Затем центральный компьютер ответит за другой компьютер.

В этой ситуации мы называем центральный компьютер сервером, а подключенные к нему компьютеры — клиентами.

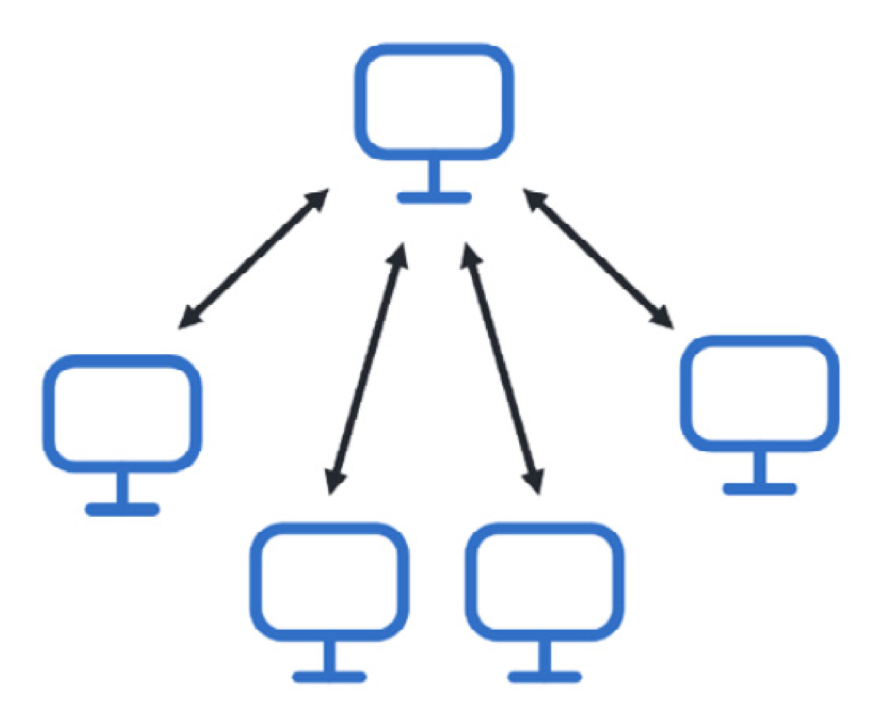


Рисунок 11.5 – В клиент-серверной сети каждый компьютер общается с сервером.

Благодаря такой архитектуре сервер может проверять всех клиентов и убеждаться, что никто из них не мошенничает.

Сетевое взаимодействие в Godot Engine

Опять же, Godot Engine поддерживает как одноранговую, так и клиент-серверную сетевую архитектуру. Чтобы облегчить себе задачу, мы выберем клиент-серверный подход. Таким образом, мы можем быть уверены, что важные части игры будут запускаться только на сервере, и нашим клиентам не придется о них беспокоиться. Например, рассмотрим подсчет очков — клиенты могут легко солгать об этом, в то время как теперь сервер будет единственным компьютером, ведущим счет.

Итак, после этого краткого введения в компьютерные сети, хотя нам ещё многое предстоит изучить, у нас достаточно знаний об их базовой структуре, чтобы приступить к реализации многопользовательского режима в нашей игре.

Изучение IP-адресов

В реальной жизни, чтобы отправить почту другому человеку, вам нужно знать адрес его дома. Для компьютерных сетей это примерно то же самое. Чтобы отправлять сообщения между компьютерами, нам нужно знать их **IP-адрес**. Это уникальный адрес, который гарантирует, что вы сможете найти любой компьютер, подключенный к Интернету.

В настоящее время используются две версии IP-адресов: **IPv4** и **IPv6**. Адрес IPv4 состоит из 4 чисел от **0** до **255**, разделённых точкой, например:

```
166.58.155.136
```

Эта версия позволяет иметь уникальный адрес для 4,3 миллиарда устройств. Но оказывается, что люди настолько продуктивны, что 4,3 миллиарда устройств уже недостаточно! В наши дни практически любое электрическое устройство может быть подключено к Интернету, даже холодильники, тостеры и часы. Вот почему мы медленно переходим на адреса IPv6, которые поддерживают 340 ундециллионов устройств. Это 340 триллионов устройств.

Адрес IPv6 выглядит так:

```
e9fd:da7d:474d:dedb:d152:dce2:1294:2560
```

В зависимости от того, как ваш компьютер подключен к Интернету, этот IP-адрес время от времени меняется, поэтому не стоит рассчитывать на то, что он останется прежним.

IP-адрес – это почтовый адрес, на который мы можем отправить письмо, но тогда нам все равно нужно знать, кому в доме

адресовано письмо. В компьютерной сети **порт** используется для адресации точного приложения в компьютере. Давайте поговорим о портах дальше.

Использование номеров портов

IP-адрес, будь то IPv4 или IPv6, указывает только, куда отправлять данные. Но у компьютеров есть много приложений, каждому из которых нужно своё собственное соединение. Итак, с момента получения данных, какому приложению мы их отправляем? Ну, каждое приложение может использовать разные порты, которые подобны разным платформам на железнодорожной станции. Хотя каждый поезд прибывает на одну и ту же станцию, они прибывают на разные платформы.

Каждое приложение может выбрать порт, который представляет собой просто число от **0** до **65535**. Однако первые 1024 зарезервированы для стандартных функций компьютера, и мы не сможем их выбрать.

Чтобы указать, на какой порт отправлять данные, мы можем добавить номер порта в конце IP-адреса после двоеточия:

```
166.58.155.136:5904
e9fd:da7d:474d:dedb:d152:dce2:1294:2560:5904
```

Здесь у нас есть адрес IPv4 и IPv6, который указывает на порт с номером **5904**.

Теперь, когда мы знаем об основных механизмах компьютерных сетей, таких как различные уровни и как работают IP-адреса, мы можем начать внедрять многопользовательский режим в нашу игру. Итак, давайте попробуем!

Настройка базового сетевого кода

В разделе *Краткий курс по компьютерным сетям* мы увидели, что хотим настроить клиент-серверную сетевую архитектуру и что мы можем использовать IP-адреса и порты для поиска компьютеров через Интернет. В этом разделе мы начнем это реализовывать.

Мы собираемся сделать так, чтобы наша многопользовательская игра работала следующим образом: каждый раз, когда вы начинаете играть, она запускает сервер в фоновом режиме. Таким образом, любой может присоединиться после того, как один человек начинает матч.

Создание клиент-серверного соединения

Если мы хотим подключить наших игроков через модель клиент-сервер, нам нужно иметь возможность настроить один компьютер как сервер, а другие – как клиенты, которые подключаются к этому серверу. Давайте начнем с написания кода.

1. В скрипте **menu.gd** добавьте вверху константу, указывающую, какой порт мы хотим использовать:

```
const PORT: int = 7890
```

2. Теперь добавьте эти две функции в конец скрипта:

```
func host_game():
   var peer = ENetMultiplayerPeer.new()
   peer.create_server(PORT)
   if peer.get_connection_status() == MultiplayerPe
      return
   multiplayer.multiplayer_peer = peer
func connect_to_game(ip_address: String):
   var peer = ENetMultiplayerPeer.new()
   peer.create_client(ip_address, PORT)
   if peer.get_connection_status() == MultiplayerPe
```

```
        return
    multiplayer.multiplayer_peer = peer
```

Функция **host_game()** будет использовать класс **ENetMultiplayerPeer** для создания нового сервера с помощью функции **create_server()**, которая определена в нём. Чтобы создать этот сервер, нам нужно только указать, на какой порт мы хотим получать данные. После этого мы проверяем, отключено ли состояние соединения; если мы не подключены, то нам нужно вернуться из функции. Мы можем проверить состояние соединения с помощью функции **get_connection_status()** на объекте **peer**.

3. Наконец, мы устанавливаем этот пир как **multiplayer_peer**, который определяется в глобальной переменной multiplayer.

   Функция **connect_to_game()** делает в основном то же самое, но создаёт клиента, используя функцию **create_client()** на объекте **ENetMultiplayerPeer peer**. Функция **create_client()** принимает IP-адрес и порт. Это, конечно, будут IP-адрес и порт сервера.

Имея эти две функции на месте, мы можем добавить ещё один пользовательский интерфейс для подключения к нужному серверу.

Добавление пользовательского интерфейса

Теперь, для меню, мы хотим иметь возможность запустить игру, которая настроит сервер или введёт IP-адрес, чтобы присоединиться к уже размещенной игре. Нам не придётся позволять игроку выбирать порт, и потому что это менее хлопотно для игрока, и потому что мы не хотим, чтобы он случайно выбрал неверный номер порта. Мы, программисты, решили, что будем использовать порт **7890**.

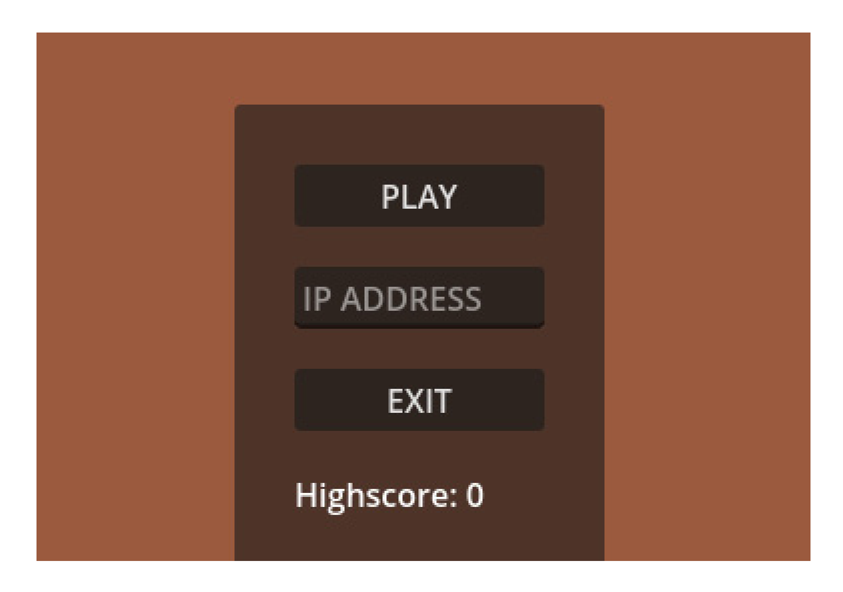


Рисунок 11.6 – Главное меню с полем ввода для указания IP-адреса

1. Откройте сцену **menu.tscn** scene.
2. Добавьте узел **LineEdit** в **VBoxContainer**, содержащий кнопки запуска игры и выхода, и переименуйте его в**IpAddressLineEdit**.
3. Поместите узел**IpAddressLineEdit** ниже узла **PlayButton**, но не как дочерний элемент.

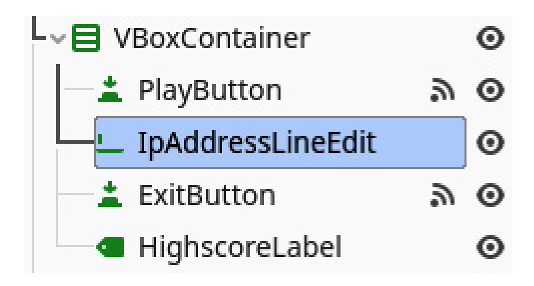


Рисунок 11.7 – Дерево сцены главного меню с добавленным IpAddressLineEdit

1. Выберите узел **IpAddressLineEdit** и впишите в поле **Placeholder Text** слова **IP ADDRESS**. Это текст-заполнитель, который будет заменён в тот момент, когда пользователь введёт что-либо в строку редактирования.

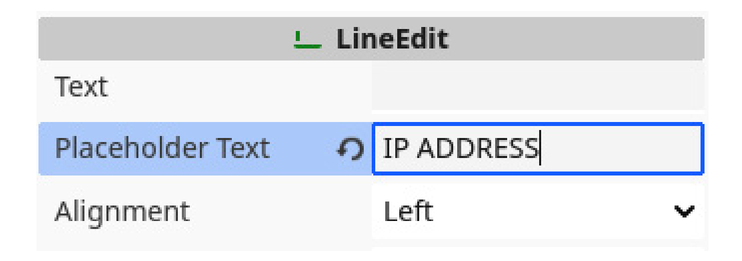


Рисунок 11.8 – Установка текста-заполнителя в узле LineEdit

1. Теперь в вверху скрипта **menu.gd** кэшируем **IpAddressLineEdit**:

   ```
   @onready var _ip_address_line_edit = $CenterContain
   ```

2. Наконец, нам нужно изменить функцию **_on_play_button_pressed()** для размещения игры (host) или подключения к ней (connect):

   ```
   func _on_play_button_pressed():
      if _ip_address_line_edit.text.is_empty():
         host_game()
      else:
         connect_to_game(_ip_address_line_edit.text)
      get_tree().change_scene_to_file("res://screens/g
   ```

Теперь у нас есть всё необходимое для настройки архитектуры клиент-сервер. Один компьютер будет сервером, а другие — клиентами. Прежде чем погрузиться в то, что нам нужно изменить в коде самой игры, чтобы, например, создать игровых персонажей для каждого присоединяющегося человека и затем убедиться, что положение каждого игрока синхронизировано между каждым компьютером, мы можем попробовать то, что мы уже создали.

Запуск нескольких экземпляров отладки одновременно

Для отладки многопользовательской игры нам нужно иметь возможность запускать нашу игру несколько раз в режиме отладки. К счастью, Godot Engine имеет удобную функцию, которая позволяет нам запускать столько экземпляров нашей игры, сколько мы хотим, одновременно.

1. Нажмите **Отладка (Debug)** в верхней строке меню.
2. В меню **Запустить несколько экземпляров (Run Multiple Instances)** выберите **Запустить 2 экземпляра (Run 2 Instances)**.

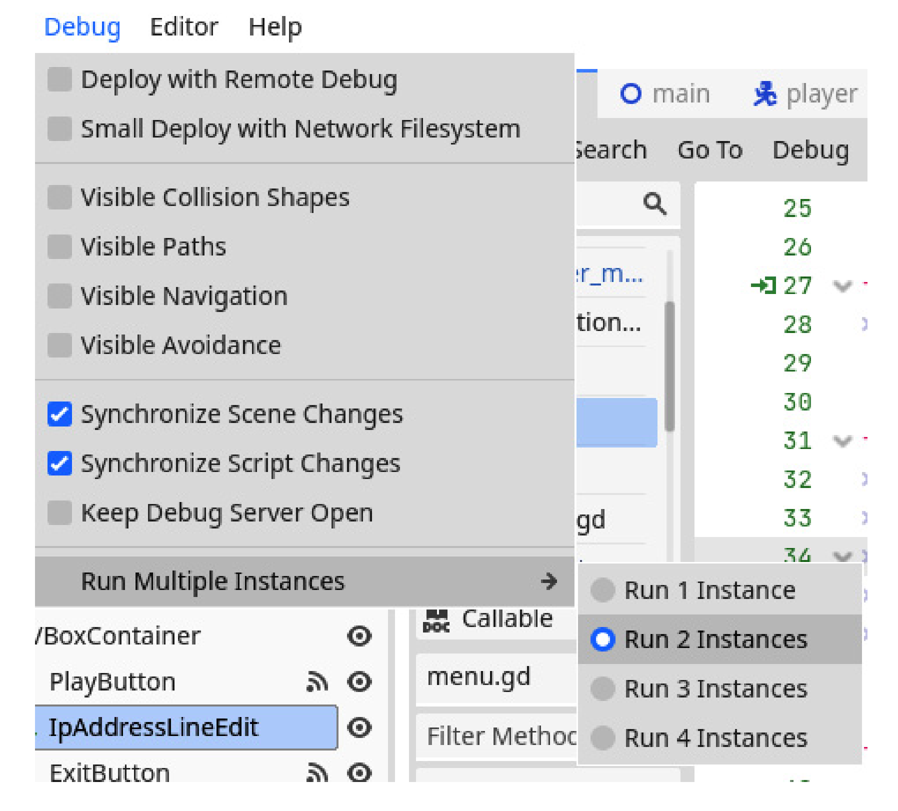


Рисунок 11.9 – В раскрывающемся меню «Отладка» мы задаём количество экземпляров, которые хотим запустить

1. Запустите проект. Это заставит одновременно появиться два экземпляра игры.
2. В одном экземпляре просто нажмите **Play**. Игра должна запуститься нормально.

3. В другом экземпляре введите **::1** в поле ввода IP-адреса и нажмите кнопку **Play**.

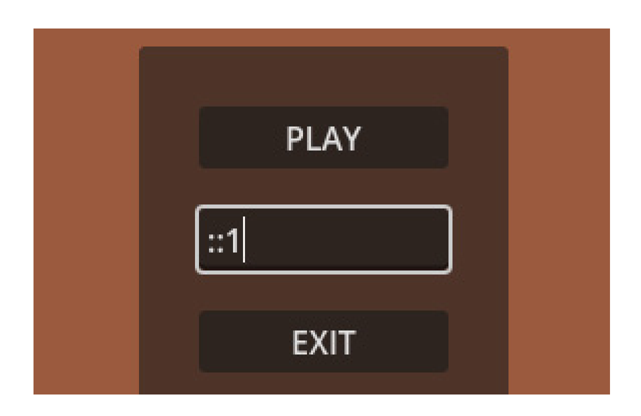


Рисунок 11.10 – Указание IP-адреса ::1 приведёт к возврату на тот же компьютер

К сожалению, вы не увидите ничего особенного. Нам всё ещё нужно написать код, учитывающий несколько игроков в нашей игре, но обычно в нижней панели **Отладка (Debug)** не должно быть ошибок.

IP-адрес локального хоста (Local host)

Есть специальный IP-адрес, который не ведёт на другой компьютер, а возвращается обратно на тот же компьютер. В формате IPv6 этот адрес **::1**, а для IPv4 это **127.0.0.1**.

Вы также увидите, что теперь на панели **Отладка Debug** есть несколько вкладок, по одной для каждого экземпляра игры. Таким образом, мы сможем отлаживать каждый по отдельности.

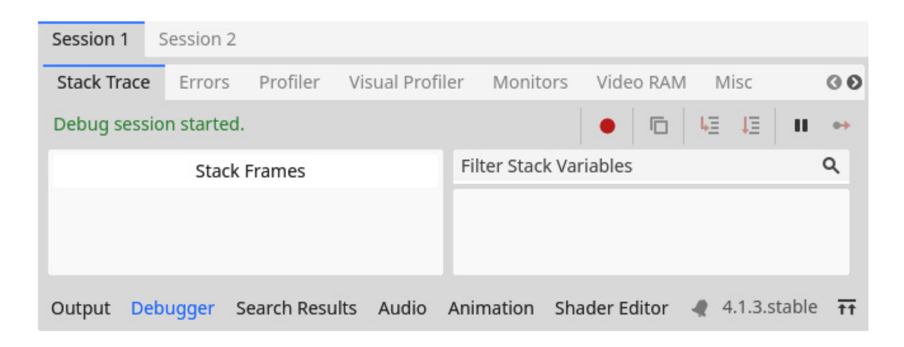


Рисунок 11.11 – При запуске нескольких экземпляров у нас также будет несколько вкладок Отладка

Теперь, когда мы можем создать сервер и подключить клиентов, давайте сделаем нашу игру многопользовательской и синхронизируем созданные сцены между обеими играми.

Синхронизация разных клиентов

До сих пор мы изучали компьютерные сети и настроили соединение между несколькими экземплярами нашей игры. Следующий шаг — изменить сцены и код в нашей игре для учета нескольких игроков. Мы хотим добиться двух вещей:

- Во-первых, если сервер создает новую сцену, например, новый снаряд, мы хотим, чтобы эта сцена была создана на каждом клиенте.
- Во-вторых, мы хотим синхронизировать значения, такие как положение каждого персонажа игрока, между всеми клиентами.

Сначала мы рассмотрим, какие узлы Godot Engine могут помочь нам достичь этих двух целей, обновляя персонажа игрока для использования в многопользовательском режиме. После этого мы также обновим спавнер сущностей, врагов, коллекционные предметы и сцены снарядов. Большинство этих изменений будут довольно небольшими.

Обновление сцены игрока для многопользовательской игры

Поскольку игрок — самая важная сущность в игре, давайте начнем с его обновления для многопользовательского режима. Таким образом, мы также сможем быстро убедиться, что все работает правильно.

Использование MultiplayerSpawner для создания сцен игроков

Для синхронизации инстансированных сцен между сервером и клиентами Godot Engine имеет узел **MultiplayerSpawner**. Он будет прослушивать сцены, которые добавляются в дерево сцен, и будет реплицировать их на каждом из других клиентов. Давайте добавим один в основную игровую сцену:

1. Откройте сцену **main.tscn**.
2. Под корневым узлом **Main** добавьте узел **MultiplayerSpawner** и назовите его **PlayerMultiplayerSpawner**, поскольку он будет создавать новых персонажей игроков.

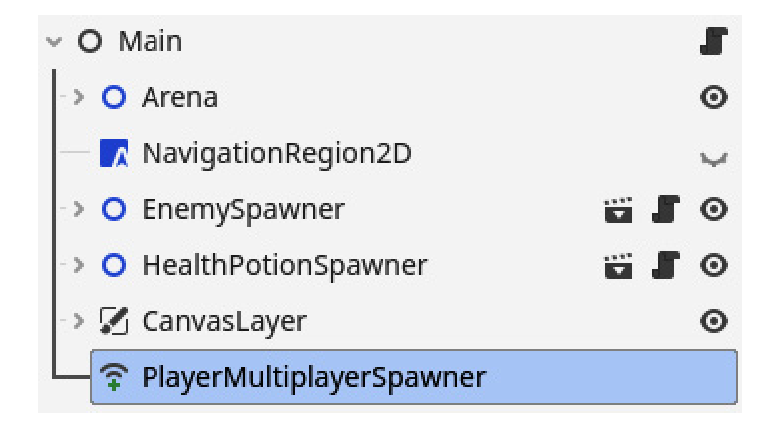


Рисунок 11.12 – Дерево сцены main.tscn с добавленным PlayerMultiplayerSpawner

1. Теперь в окне инспектора **PlayerMultiplayerSpawner**

нажмите **Добавить элемент (Add Element)** в **Auto Spawn List** и перетащите сцену **player.tscn** в этот элемент..

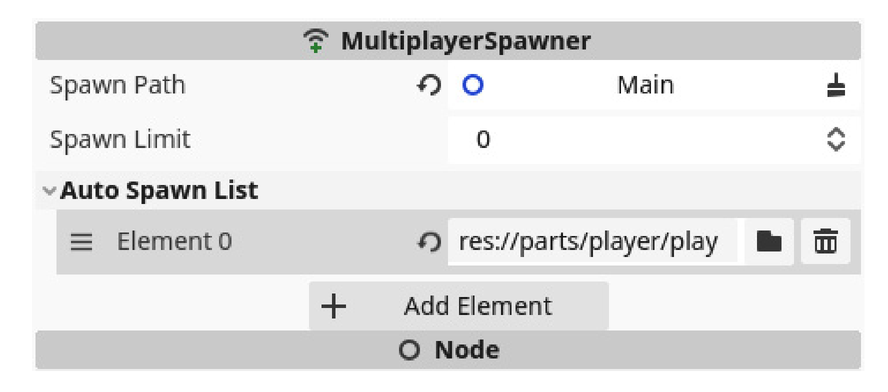


Рисунок 11.13 – Добавление сцены player.tscn как элемента в PlayerMultiplayerSpawner

1. Теперь, чтобы указать позиции, в которых могут появляться наши игроки, добавьте **Node2D** с именем **PlayerStartPositions**, и расположите его под узлом **Main**. Добавьте к нему узлы **Marker2D** в тех позициях, где мы планируем возможные появления игроков.



Рисунок 11.14 – Узел PlayerStartPositions с дочерними узлами Marker2D для появления игроков

1. В скрипте **main.gd** мы кэшируем сцену игрока в переменной экспорта. Поэтому добавьте следующую строку кода вверху и перетащите сцену **player.tscn** в эту переменную экспорта в инспекторе:

```
@export var player_scene: PackedScene
```

2. Кроме того, кэшируйте узел **PlayerMultiplayerSpawner** в переменной с именем **_player_multiplayer_spawner** и **PlayerStartPositions** в переменной с именем **_player_start_positions**:

```
@onready var _player_multiplayer_spawner: Multiplay
@onready var _player_start_positions: Node2D = $Pla
```

3. Мы также добавим переменную в верхней части скрипта, которая определит, в какой позиции мы будем создавать следующего игрока. С помощью этой переменной мы выберем, какой **Marker2D** использовать в качестве местоположения для создания каждого игрока:

```
var _player_spawn_index: int = 0
```

4. Теперь добавим две функции для создания новых игроков в скрипт **main.gd**:

```
func add_player(id: int):
   _player_multiplayer_spawner.spawn(id)
func spawn_player(id: int):
   var player: Player = player_scene.instantiate()
   player.multiplayer_id = id
   player.died.connect(_on_player_died)
   var spawn_marker: Marker2D = _player_start_posit
   player.position = spawn_marker.position
   _player_spawn_index = (_player_spawn_index + 1)
   return player
```

5. Чтобы использовать эти функции, мы добавим функцию **_ready()** в скрипт **main.gd**:

```
func _ready():
   _player_multiplayer_spawner.spawn_function = spa
```

```
    if multiplayer.is_server():
        multiplayer.peer_connected.connect(add_player
        add_player(1)
```

6. И последнее, но очень важное — удалите узел **Player**, который уже находится в сцене **main.tscn**. Мы делаем это, потому что мы будем создавать каждого персонажа игрока из кода, и им не нужно, чтобы узел уже был там.

В функции **add_player()** мы просто просим **_player_multiplayer_spawner** создать новый экземпляр сцены игрока.

Затем в функции **spawn_player()**, которая будет использоваться **PlayerMultiplayerSpawner** для создания новых сцен **Player**, мы создаём новую сцену игрока и устанавливаем её свойство **multiplayer_id** на ID, который мы получили в качестве параметра. Этот ID используется для определения того, какой клиент владеет этим конкретным узлом Player. Мы используем его в следующем разделе. После этого мы должны вернуть новый экземпляр сцену игрока (player), чтобы **PlayerMultiplayerSpawner** мог обработать всё остальное за нас.

Мы используем переменную **_player_spawn_index** для выбора **Marker2D** в **PlayerStartPositions**. После появления каждого игрока мы увеличиваем эту переменную на **1** и обеспечиваем её возврат в цикл с помощью оператора **%**. Это гарантирует, что мы не будем создавать игроков друг над другом.

В функции **_ready()** мы сначала устанавливаем **spawn_function** для **_player_multiplayer_spawner** как функцию **spawn_player()**, которую мы определили. Таким образом, многопользовательский спаунер знает, как создавать новые экземпляры сцены игрока.

Затем мы проверяем объект **multiplayer**, выполняется ли этот код на сервере, используя **multiplayer.is_server()**. Функция **is_server()** возвращает **true**, если код выполняется на сервере.

Если мы работаем на сервере, то делаем следующее:

```
multiplayer.peer_connected.connect(add_player)
```

Сигнал **peer_connected —** это сигнал, который выдаётся объектом **multiplayer**, когда новый клиент (новый peer) подключается к серверу. Вместо подключения через редактор, как мы делали раньше для определения того, находится ли игрок близко к предметам коллекционирования, мы напрямую вызываем функцию **connect()** для этого сигнала и передаем функцию, которую хотим выполнить, когда игрок подключается к серверу, то есть функцию **add_player()**.

После подключения к сигналу **peer_connected** мы вызываем функцию **add_player()** с идентификатором **1**, который является идентификатором по умолчанию для сервера.

На данный момент мы не сможем запустить игру, для начала нам нужно обновить сцену игрока.

Обновление кода игрока для многопользовательской игры

Если вы попробуете запустить игру с несколькими экземплярами в конце последнего раздела, вы заметите, что некоторые вещи работают не так, например, на каждом клиенте по отдельности вы управляете обоими игроками одновременно.

Такое поведение происходит, потому что, хотя мы создаем игрока на клиенте, весь код всё время запускается на каждом отдельном клиенте. Мы должны указать, что код движения для каждого персонажа игрока должен запускаться только на клиенте, связанном с этим персонажем игрока, а не все сразу на всех клиентах. После этого мы должны синхронизировать позицию с сервером.

Мы сделаем это, установив **многопользовательские полномочия (multiplayer authority)** узла персонажа игрока. Эти полномочия «владеют» этим узлом и решают, как он себя ведёт.

Итак, давайте изменим наш код, чтобы персонажи игроков

работали правильно:

1. Во-первых, добавьте переменную **multiplayer_id** , которую мы использовали в предыдущем разделе, где-нибудь в верхней части скрипта **player.gd**:

```
var multiplayer_id: int
```

2. Добавьте функцию **_enter_tree()**. Эта функция является функцией жизненного цикла, которая вызывается, когда узел входит в дерево, прямо перед функцией **_ready()**. В этой функции мы устанавливаем полномочия многопользовательского режима для клиента, имеющего тот же идентификатор (ID) что и **multiplayer_id** этого узла игрока:

```
func _enter_tree():
   set_multiplayer_authority(multiplayer_id)
```

3. Кэшируйте узел **CameraPosition** в верхней части скрипта:

```
@onready var _camera_position: Node2D = $CameraPosi
```

4. Теперь обновите функцию **_ready()** следующим образом:

```
func _ready():
   update_health_label()
   if not multiplayer.is_server():
      _shoot_timer.stop()
   if not is_multiplayer_authority():
      _camera_position.queue_free()
      set_physics_process(false))
```

На *шаге 2* мы устанавливаем полномочия многопользовательского режима для узла, что означает, что мы определяем, какой клиент является владельцем этого узла. Для большинства узлов в многопользовательском режиме

владельцем должен быть сервер. Но персонаж игрока настолько важен для каждого клиента, что мы даём полномочия каждого из них соответствующему клиенту.

После этого мы используем **multiplayer.is_server()** для остановки **_shoot_timer**, когда мы работаем не на сервере. Таким образом, мы гарантируем, что снаряды будут созданы только на стороне сервера и реплицированы на всех клиентов оттуда.

Далее мы используем **is_multiplayer_authority()** для проверки, есть ли у нас полномочия на этот конкретный узел игрока. Если нет, мы освобождаем **_camera_position**. Нам не нужны несколько камер, только та, которая используется для отслеживания игрока, которого мы хотим видеть, и мы также отключаем функцию **_physics_process()**. Только клиент, владеющий этим узлом, должен будет вычислить позицию этого игрока и затем сообщить серверу, где находится игрок.

Отключение функций _process() и _physics_process()

По умолчанию функции **_process()** и **_physics_process()** вызываются для каждого кадра и физического кадра соответственно. Однако мы можем включить или отключить их вручную, вызвав **set_process()** и **set_physics_process()** вместе с булевым значением, которое указывает, должны ли они запускаться или нет.

После всего этого вы можете запустить несколько экземпляров игры, как мы видели в разделе *Запуск нескольких экземпляров отладки одновременно*, и вы должны увидеть появление второго игрока! Каждый игрок может нормально двигаться, но их позиции, к сожалению, пока не синхронизированы. Мы сделаем это далее.

Синхронизация позиций и здоровья игроков

Мы можем создавать сцены на разных клиентах и определять, на каком клиенте должны выполняться определенные фрагменты кода. Последняя часть головоломки — синхронизировать определённые переменные, такие как

положение и здоровье наших игроков. К счастью, это на самом деле очень легко сделать с помощью узла **MultiplayerSynchronizer**. Мы собираемся использовать два из них, один для положения и один для здоровья. Хотя один синхронизатор может синхронизировать несколько переменных, мы хотим, чтобы положение управлялось каждым клиентом индивидуально, а здоровье — сервером:

1. В сцене **player.tscn** добавьте два узла **MultiplayerSynchronizer** под корневым узлом **Player**. Назовите один из них **PositionMultiplayerSynchronizer**, а другой — **HealthMultiplayerSynchronizer**.

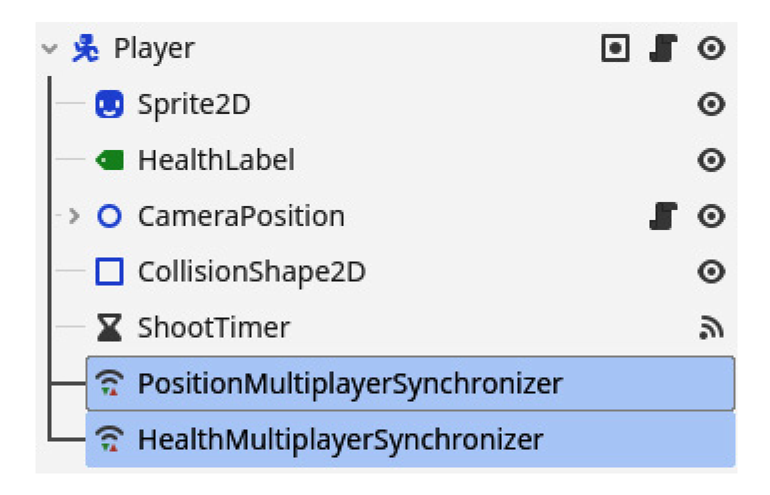


Рисунок 11.15 – Дерево сцены player.tscn после добавления двух узлов MultiplayerSynchronizer

1. Выберите **PositionMultiplayerSynchronizer**, и в нижней части редактора должна появиться новая панель.

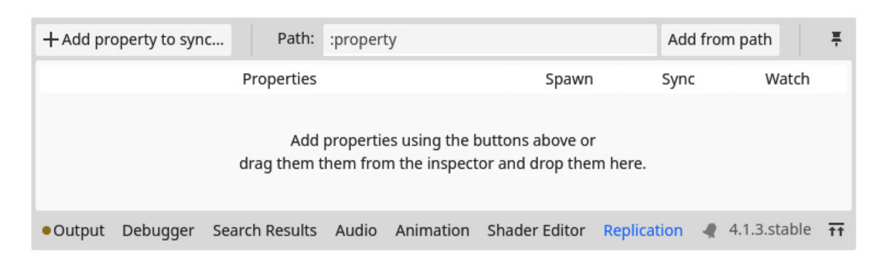


Рисунок 11.16 – Панель репликации (Replication panel), которая открывается при выборе MultiplayerSynchronizer

1. Здесь нажмите + **Добавить свойство для синхронизации (+ Add property to synchronize)**.
2. Выберите узел **Player** и нажмите **OK**.

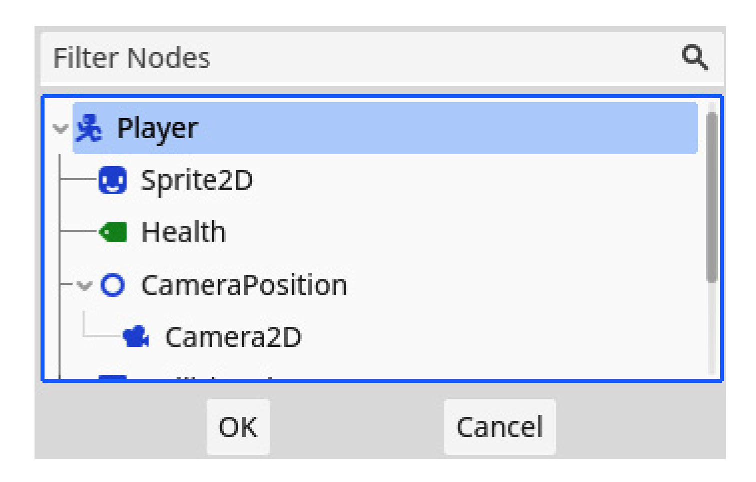


Рисунок 11.17 – Выберите узел Player, чтобы синхронизировать одно из его значений

1. Теперь найдите свойство **position** и нажмите **Открыть (Open)**.

Search:

Matches:

input_pickable

Node2D

position

rotation

Description:

position: Position, relative to the node's parent.

Open Cancel

Рисунок 11.18 – Выберите свойство position для синхронизации его значения

1. Повторите *шаги 2 — 5* ещё раз , но на этот раз добавьте свойство **health** в **HealthMultiplayerSynchronizer**.

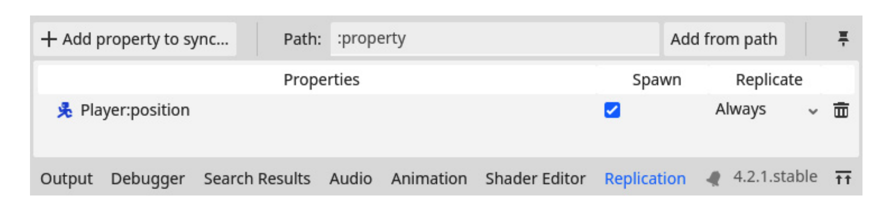


Рисунок 11.19 – Панель репликации (Replication), отслеживающая значение position узла Player

1. Теперь обновим функцию **_enter_tree()** игрока, чтобы предоставить серверу полномочия многопользовательского режима **HealthMultiplayerSpawner**:

```
func _enter_tree() -> void:
    set_multiplayer_authority(multiplayer_id)
    $HealthMultiplayerSynchronizer.set_multiplayer_a
```

Помните, что идентификатор (ID) многопользовательского режима сервера всегда равен **1**. Поэтому, чтобы предоставить серверу полномочия (authority), мы устанавливаем многопользовательские полномочия **HealthMultiplayerSynchronizer** на **1**.

Это всё, что нам нужно сделать для синхронизации значений между разными клиентами. **MultiplayerSynchronizer** просто отслеживает их для нас.

Запуск двух экземпляров игры и их соединение в конечном итоге показывает, что если мы перемещаем одного персонажа игрока в одном клиенте, это также перемещает этого персонажа игрока в другом клиенте.

Теперь, когда мы обновили самую сложную сцену для многопользовательского режима, сцену игрока, у нас есть все знания, чтобы сделать то же самое для оставшихся сцен. Давайте займёмся этим, чтобы в конце у нас была полноценная многопользовательская игра!

Синхронизация EntitySpawner

Чтобы сцены врагов и зелий здоровья появлялись на каждом клиенте тогда, когда этого хочет создатель (spawner) сущностей, нам придется внести несколько небольших изменений в сцену **EnitySpawner**:

1. В сцене **entity_spawner.tscn** добавьте узел **MultiplayerSpawner**.

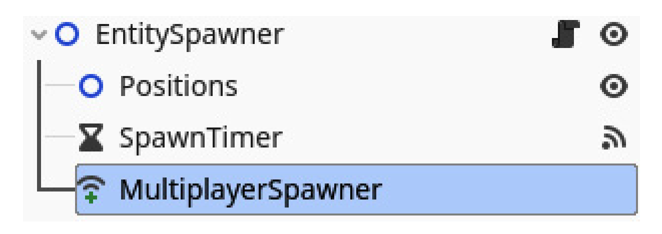


Рисунок 11.20 – Дерево сцены EntitySpawner после добавления MultiplayerSpawner

1. В скрипте **entity_spawner.gd** кэшируйте узел **MultiplayerSpawner**:

```
@onready var _multiplayer_spawner = $MultiplayerSpa
```

2. Затем в функции **_ready()** давайте добавим сцену, которую использует этот спавнер, к узлу **MultiplayerSpawner** и запустим таймер только если мы работаем на сервере. Это гарантирует, что не каждый клиент спавнит новые сущности, а только сервер:

```
func _ready():
   _multiplayer_spawner.add_spawnable_scene(entity_
   if multiplayer.is_server():
      start_timer()
```

3. Последнее, что нам нужно сделать, это изменить способ добавления **new_entity** на сцену. Итак, измените **add_child(new_entity)** на следующее:

```
add_child(new_entity, true)
```

На *шаге 3* мы добавляем устанавливаемую сцену в узел **MultiplayerSpawner**Это очень удобно, так как теперь мы можем добавлять любую сцену на лету.

На *шаге 4* мы передаём логическое значение **true** в качестве второго параметра функции **add_child()** рядом с узлом, который мы хотим добавить в дерево сцены. Это означает, что мы хотим использовать понятные человеку имена для каждого узла. Если мы не укажем значение **true**, движок выберет имя для узла. Эти имена выглядят как **@Node2D@2**. Это зарезервированные имена, которые нельзя синхронизировать с помощью узла **MultiplayerSpawner**. Если мы устанавливаем это логическое значение в **true**, каждый новый экземпляр получает

красивое имя, например **Enemy2**, **Enemy3** и т.д. В многопользовательском сценарии это важно для сервера, чтобы правильно синхронизировать сцены и значения между ними.

Теперь, когда мы можем синхронизировать создаваемые сущности врагов и предметы коллекционирования между клиентами, давайте синхронизируем и их поведение.

Синхронизация врагов и предметов коллекционирования

Как врагов, так и коллекционные предметы заставить работать в многопользовательском режиме довольно легко и просто:

1. Добавьте **MultiplayerSynchronizer** в сцены **enemy.tscn** и **collectible.tscn**.

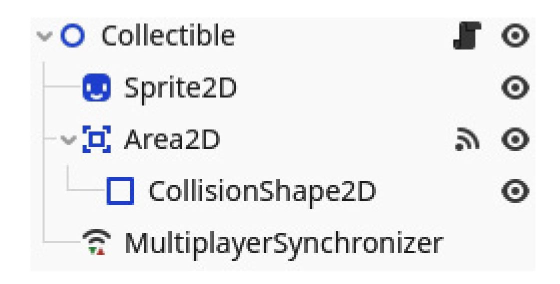


Рисунок 11.21 – Дерево сцены коллекционного предмета после добавления MultiplayerSynchronizer

1. Теперь добавьте свойство **position** корневого узла в меню **Репликация (Replication)** внизу.

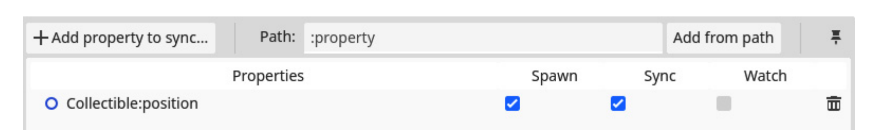


Рисунок 11.22 – Панель репликации, отслеживающая положение узла Collectible

Это всё для **Collectible**, а для **Enemy** нам нужно сделать ещё несколько последних вещей в коде:

1. Кэшируйте **PlayerDetectionArea** в верхней части скрипта **enemy.gd**:

```
@onready var _player_detection_area: Area2D = $Play
```

2. Теперь обновите функцию **_ready()** следующим образом:

```
func _ready():
   if not multiplayer.is_server():
      set_physics_process(false)
      _player_detection_area.monitoring = false
      return
   var player_nodes: Array = get_tree().get_nodes_i
   if not player_nodes.is_empty():
      target = player_nodes.pick_random()
```

Первое, что мы делаем в функции **_ready()** противника, это отключаем функции **_physics_process()** и **_player_detection_area**, если мы не запускаем их с сервера. Это гарантирует, что враги полностью контролируются сервером.

Узлы **Area2D** имеют свойство **monitoring**, которое прекращает поиск столкновений с другими областями или телами при установке значения **false**. Это то, что мы используем здесь для отключения **_player_detection_area** на всех клиентах, кроме сервера.

Наконец, мы хотим иметь возможность нацеливаться на любого игрока в игре, поэтому мы меняем способ нацеливания на игрока. Функция **pick_random()** в массиве выберет любой элемент в этом массиве случайным образом и вернёт его. Это идеально подходит для выбора случайного игрока в пределах сцены!

Давайте теперь рассмотрим, как можно синхронизировать снаряды.

Синхронизация снаряда

Последняя сцена, которую нам нужно синхронизировать между несколькими клиентами, — это сцена со снарядами. Итак, сделаем это, выполнив следующие шаги:

1. В сцену **projectile.tscn** добавьте **MultiplayerSynchronizer**.

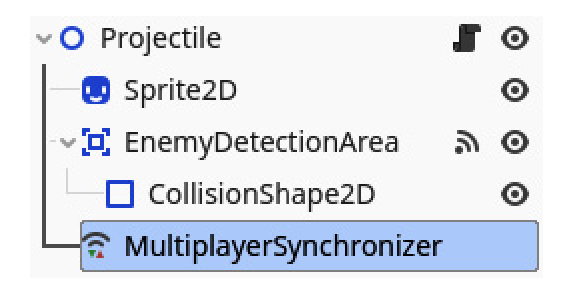


Рисунок 11.23 – Дерево сцены Projectile после добавления MultiplayerSynchronizer

1. На этот раз синхронизируйте свойства **position** и **rotation**.

+ Add property to sync... | Path: :property | Add from path

Properties	Spawn	Sync	Watch
Projectile:position	☑	☑	☐
Projectile:rotation	☑	☑	☐

Рисунок 11.24 – Панель репликации, отслеживающая position и rotation узла Projectile

Кэшируйте **EnemyDetectionArea** в верхней части скрипта **projectile.gd**:

```
@onready var _enemy_detection_area: Area2D = $EnemyDetec
```

1. Теперь добавьте функцию **_ready()** следующим образом:

```
func _ready():
    if not multiplayer.is_server():
        set_physics_process(false)
        _enemy_detection_area.monitoring = false
```

2. Нам нужно изменить способ добавления снаряда на сцену в скрипте **player.gd** с **get_parent().add_child(new_projectile)** на следующий:

```
get_parent().add_child(new_projectile, true)
```

Важное примечание

Помните, что последний параметр функции **add_child()** — это логическое значение, которое определяет, должно ли имя нового узла быть удобочитаемым.

1. Наконец, нам нужно убедиться, что сцена **projectile.tscn** реплицируется в основной сцене, как мы это сделали для сцены **player.tscn**. Добавьте узел **MultiplayerSpawner** в **main.tscn**, назовите его **ProjectileMultiplayerSpawner** и добавьте **projectile.tscn** в **Auto Spawn List**.

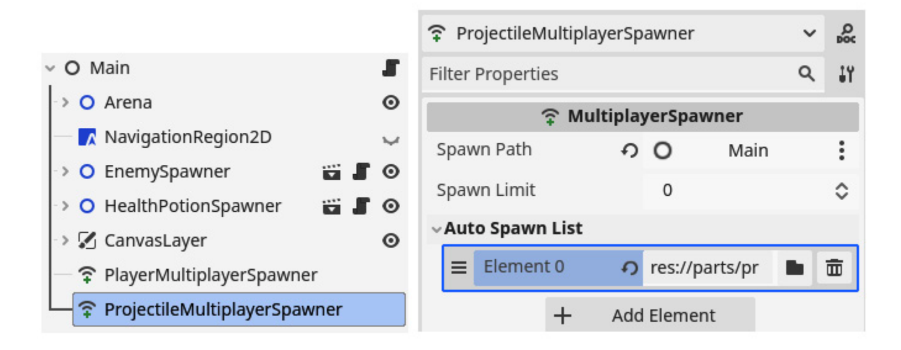


Рисунок 11.25 – Основная сцена с ProjectilMultiplayerSpawner, где установлена сцена projectile.tscn

Вот и всё для сцены **Projectile** и, следовательно, для всех сцен, важных для самой игры! Теперь вы можете запустить несколько экземпляров игры, и всё в игре должно быть синхронизировано. Последнее, что нам нужно рассмотреть, это синхронизация таймеров в игре и меню окончания игры для обоих игроков.

Исправление таймера и

завершение игры

Последнее, что нам нужно настроить для многопользовательской игры, это таймер, который отсчитывает время нашего бега и конец игры, останавливая спавнеры сущностей и показывая меню окончания игры. Итак, давайте совершим это последнее усилие.

Синхронизация таймера

Чтобы синхронизировать таймер счета, нам просто нужно сделать следующие три вещи:

1. Добавьте **MultiplayerSynchronizer** в сцену **main.tscn**.

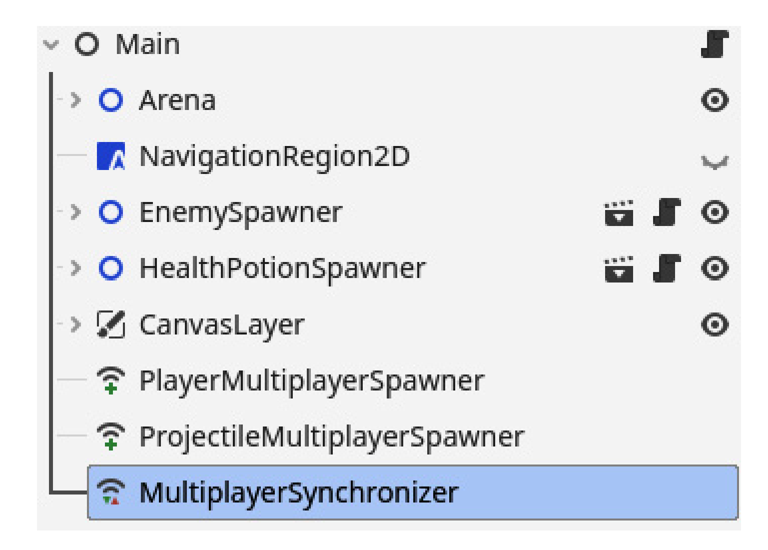


Рисунок 11.26 – Дерево сцены Main после добавления MultiplayerSynchronizer

1. Синхронизируйте свойство **_time** узла **Main**.

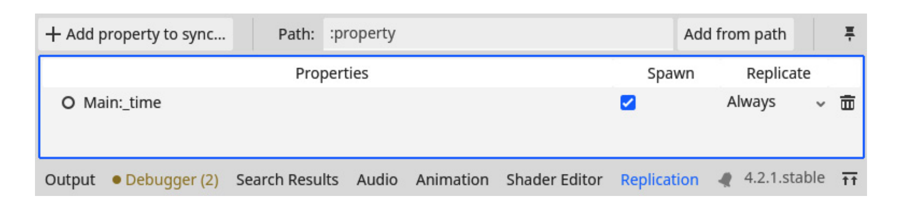


Рисунок 11.27 – Панель репликации, отслеживающая свойство _time узла Main

1. Теперь отключите функцию **_process()** из функции **_ready()**, если она запущена не на сервере:

```
func _ready():
    # Other code
    if not multiplayer.is_server():
        set_process(false)
```

Это всё, что нам нужно сделать, чтобы синхронизировать таймер на всех клиентах.

Синхронизация конца игры

Чтобы быть уверенным, что когда игра закончится, она закончится для всех клиентов, сделаем следующее:

1. В скрипте **main.gd** подключимся к сигналу **died** каждого персонажа игрока в функции **add_player()**:

```
func add_player(id: int):
    var player: Player = player_scene.instantiate()
    player.name = str(id)
    add_child(player)
    player.died.connect(_on_player_died)
```

2. Теперь измените функцию **_on_player_died()** и добавьте новую функцию **end_game()**:

```
func _on_player_died() -> void:
```

```
    end_game.rpc()
@rpc("authority", "reliable", "call_local")
func end_game():
    _game_over_menu.show()
    _enemy_spawner.stop_timer()
    _health_potion_spawner.stop_timer()
    set_process(false)
    Highscore.set_new_highscore(_time)
```

3. Затем в скрипте **menu.gd** измените функцию **_ready()** на следующую:

```
func _ready():
    _highscore_label.text = "Рекорд: " + str(Highsco
    if multiplayer.has_multiplayer_peer():
        multiplayer.multiplayer_peer.close()
```

На первом этапе мы просто подключаемся к сигналу **died** каждого игрока с помощью кода.

Важное примечание

Обратите внимание, что к сигналу **died** подключается только сервер , поскольку именно сервер управляет игровым циклом.

На втором этапе мы делаем что-то очень интересное. Мы вызываем функцию **end_game()** через **RPC**, что означает, что мы вызываем её на каждом клиенте одновременно!

Важное примечание

Удаленный вызов процедур (Remote procedure call) (**RPC**) — это протокол, который делает функции напрямую вызываемыми через разных клиентов. Это позволяет легко выполнять один и тот же код на всех подключенных экземплярах игры одновременно.

Вы можете видеть, что мы используем аннотацию **@rpc** прямо перед функцией **end_game()**. Это указывает, что мы хотим, чтобы эта функция обрабатывалась при вызове на каждом

клиенте одновременно. Строки, которые мы ей передаём, означают следующее:

- "**authority**": только тот, у кого есть полномочия, в данном случае сервер, может вызвать эту функцию.
- "**reliable**": мы хотим, чтобы эта команда была надёжно отправлена по сети с использованием TCP.
- "**call_local**": эта функция при вызове должна быть выполнена на всех клиентах, включая того, который её вызвал.

Это означает, что меню окончания игры будет отображаться на каждом клиенте с момента смерти одного из игроков.

На третьем шаге мы просто закрываем многопользовательское соединение, если оно есть, и открываем главное меню. Таким образом, мы гарантируем, что не останемся подключенными, если уже не играем.

Теперь, когда вся игра готова к многопользовательскому режиму, давайте начнем с её фактического запуска на нескольких компьютерах одновременно!

Запуск игры на нескольких компьютерах

До этого момента мы запускали несколько экземпляров нашей игры на одной машине. Но сила многопользовательской игры заключается в игре с несколькими людьми на нескольких машинах.

В этом разделе мы начнём с отображения IP-адреса сервера на экране, а затем рассмотрим, как можно запустить экземпляр отладки на нескольких компьютерах одновременно, чтобы они могли подключиться.

Отображение IP-адреса сервера

Мы использовали **::1** в качестве IP-адреса, который возвращается к тому же компьютеру, чтобы мы могли отлаживать нашу игру. Однако, прежде чем мы сможем подключиться к другому компьютеру по сети, нам нужно узнать его реальный IP-адрес. Для этого мы покажем IP-адрес сервера на экране, когда он будет размещать игру.

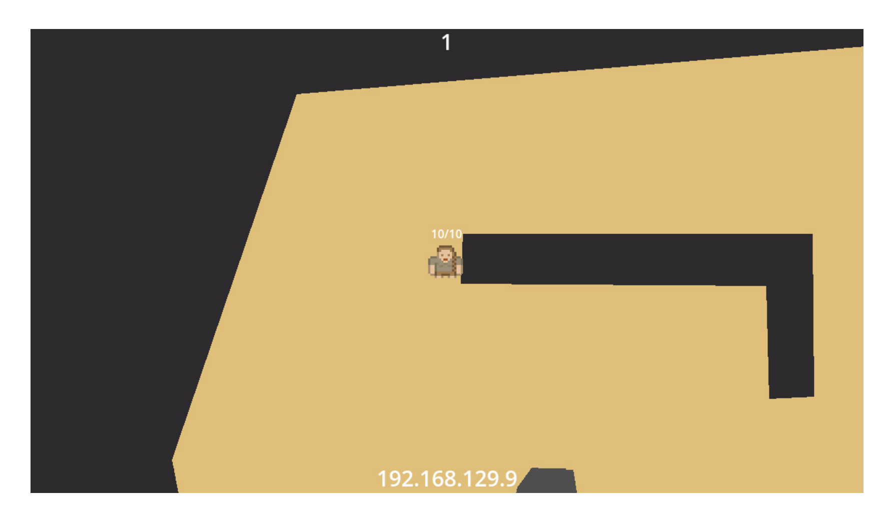


Рисунок 11.28 – Сервер имеет IP-адрес, отображаемый в нижней части экрана для подключения

На *Рисунке 11.28* вы можете видеть, что мы хотим показать IP-адрес в нижней части экрана. Давайте приступим к этому:

1. В сцене **main.tscn** добавьте **CenterContainer** с **Label** в качестве дочернего элемента, как мы это делали для таймера. Дайте им имена, как на *Рисунке 11.29*.

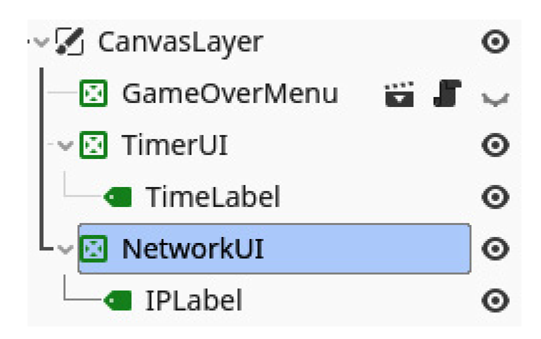


Рисунок 11.29 – Узел CanvasLayer с NetworkUI

1. Теперь в скрипте **main.gd** кэшируем **IPLabel** вверху:

```
@onready var _ip_label = $CanvasLayer/NetworkUI/IPL
```

2. Далее добавьте эту функцию, которая показывает локальный IP-адрес:

```
func show_local_ip_address():
   var addresses = []
   for ip in IP.get_local_addresses():
      if ip.begins_with("10.") or ip.begins_with("1
         addresses.push_back(ip)
   if not addresses.is_empty():
      _ip_label.text = addresses[0]
```

3. Теперь вызовите эту функцию в функции **_ready()**, но только если мы работаем на сервере:

```
func _ready():
   if multiplayer.is_server():
      show_local_ip_address()
   # …
```

Не беспокойтесь слишком сильно о реализации функции **show_local_ip_address()**. Основа её в том, что она будет искать локальный IP-адрес, сканируя все сетевые адреса текущего компьютера и сохраняя те, которые начинаются с "**10.**", "**172.16.**" или "**192.168.**", которые являются известными началами для локальных IP-адресов. Причины, по которым это работает, немного сложны и выходят за рамки этой книги.

Теперь, когда мы знаем, какой IP-адрес у сервера, давайте посмотрим, как на самом деле можно настроить всё для соединения двух компьютеров.

Подключение с другого компьютера

Главное предостережение на данный момент, о котором мы уже упоминали во введении к главе, заключается в том, что мы не сможем играть через настоящий всемирный интернет. Это связано с многочисленными причинами безопасности; вы бы не хотели, чтобы незнакомцы имели прямой доступ к вашему компьютеру. Однако мы сможем играть в одной и той же локальной сети. Это означает, что два компьютера, подключенные к одному маршрутизатору, одной и той же сети Wi-Fi и т.д., смогут подключаться друг к другу в игре! Всё, что нам нужно будет сделать, это следующее:

1. Перенесите весь проект Godot на другой компьютер. Вы можете сделать это любым удобным для вас способом. С помощью USB, с помощью онлайн-платформы, такой как Dropbox, Google Drive или любого другого средства передачи файлов.
2. Убедитесь, что оба компьютера подключены к одной локальной сети.
3. Откройте проект в той же версии Godot Engine, которую вы используете.
4. Запустите отладочный экземпляр игры на каждом компьютере.
5. Нажмите **Играть** на одном компьютере, сделав его сервером. Используйте IP-адрес, отображаемый сервером, для подключения к другим клиентам.

Теперь вы сможете играть вместе по сети!

Вот и всё, что касается соединения нескольких компьютеров. Перейдем к краткому изложению главы, но сначала вот несколько дополнительных упражнений для закрепления наших знаний.

Дополнительные упражнения – Заточка топора

1. Когда игра закончилась, мы получили меню с кнопкой **Начать заново (Retry)**, но эта кнопка **Начать заново**

(Retry) не работает должным образом в многопользовательском режиме. Можете ли вы найти способ, чтобы мы правильно начинали новую игру, когда все игроки нажимают эту кнопку? Сервер должен будет использовать функцию **add_player()** для каждой пары, которая подключена в функции **_ready()** скрипта **main.gd**. Вы можете получить список всех идентификаторов пиров с помощью **multiplayer.get_peers()**.

Итоги

Удовольствие от видеоигр заключается в том, чтобы делиться опытом, и нет ничего проще, чем играть вместе!

В этой главе мы начали с прохождения экспресс-курса по компьютерным сетям, где мы узнали основы работы компьютерных сетей, таких как Интернет. После этого мы начали внедрять многопользовательский режим в нашу собственную игру, используя узлы **MultiplayerSpawner** и **MultiplayerSynchronizer**. Наконец, мы попробовали сыграть в игру по реальной сети.

Эта глава знаменует собой конец *Главы 2* книги, где мы сосредоточились на изучении того, как разрабатывать нашу игру и делать это. Начиная со следующей главы, мы узнаем, как экспортировать игру, немного углубимся в более продвинутые темы программирования и увидим, как можно сохранить или загрузить игру.

Опрос

- В чём разница между TCP и UDP?
- Если взять в качестве примера жилой дом с квартирами, где номер порта — это номер квартиры, то что представляет собой IP-адрес?
- Для чего мы использовали **MultiplayerSpawner**?
- Для чего мы использовали **MultiplayerSynchronizer**?
- Какую функцию мы бы использовали, чтобы проверить,

запущен ли текущий скрипт на сервере?

Часть 3: Углубление наших знаний

Научившись программировать и создав свою собственную игру с нуля, теперь вы сделаете шаг назад и изучите некоторые более продвинутые методы программирования и разработки игр.

К концу этой заключительной части вы экспортируете и распространите свою игру на различные платформы в Интернете, чтобы каждый мог играть в нее из своего браузера. Вы также узнаете более продвинутые концепции ООП и различные шаблоны программирования, которые помогут вам в ваших будущих игровых проектах. Будет рассмотрена даже файловая система, чтобы вы могли сохранять и загружать данные. Последняя глава проведет вас через следующие шаги, к каким ресурсам вы можете обратиться, чтобы узнать больше, и как присоединиться к сообществу разработчиков игр.

Эта часть состоит из следующих глав:

- *Глава 12*, *Экспорт на несколько платформ*
- *Глава 13*, *Продолжение и продвинутые темы ООП*
- *Глава 14*, *Расширенные шаблоны программирования*
- *Глава 15*, *Использование файловой системы*
- *Глава 16*, *Что дальше?*

12

Экспорт на несколько платформ

После создания игры мы должны передать её в руки игроков. В конце концов, игры предназначены для того, чтобы в них играть! В старые времена это означало записывать тысячи или миллионы компакт-дисков и распространять их по всему миру в физических игровых магазинах в надежде, что люди их купят. Это стоило огромных денег и рабочей силы. Крупные студии часто получали только 10% прибыли, потому что остальное тратилось на покупку физических компакт-дисков и выплату дистрибьюторских и магазинных отчислений. Даже если вы разработали успешную хитовую игру, первоначальные инвестиции в её распространение могли остановить её на пути.

С развитием Интернета и игровых платформ, таких как **Steam** и **Itch.io**, распространение стало намного дешевле, иногда даже бесплатным и проще, практически без первоначальных вложений.

В этой главе мы узнаем всё об экспорте производственных сборок нашей игры и даже о том, как загрузить её на Itch.io (если захотите).

В этой главе мы рассмотрим следующие основные темы:

- Экспорт игры для Windows, Mac и Linux
- Загружаем нашу игру на Itch.io
- Экспорт нашей игры на другие платформы

Технические требования

Как и в случае с каждой главой, окончательный код можно

найти в репозитории GitHub в подпапке для этой главы: https://github.com/PacktPublishing/Learning-GDScript-by-Developing-a-Game-with-Godot-4/tree/main/chapter12.

Экспорт для Windows, Mac и Linux

Экспорт — это процесс, в котором мы берём игру, которую мы разрабатываем, и делаем её исполняемой вне Godot. Да, мы можем запустить нашу игру из редактора движка в режиме отладки, но в идеале мы не хотим делиться кодом нашей игры с игроками. Мы также не хотим, чтобы нашим игрокам сначала пришлось загружать весь движок с его редактором, а затем нашу игру.

При экспорте для компьютеров нам придется сделать отдельный экспорт для каждой отдельной операционной системы, например Windows, macOS и Linux. Операционная система — это программное обеспечение, которое управляет всем оборудованием и общими функциями компьютера.

Независимо от того, какую из **операционных систем** (**ОС**) вы используете, Windows, Mac или Linux, Godot позволяет нам экспортировать игру на любую другую платформу. Таким образом, мы можем легко экспортировать для всех наших пользователей, независимо от того, какую платформу они предпочитают.

Прежде чем мы сможем экспортировать данные на любую платформу, нам необходимо загрузить шаблон экспорта.

Загрузка шаблона экспорта

Чтобы экспортировать что-либо, движок Godot использует шаблон, который сообщает ему, как на самом деле экспортировать нашу игру, который уникален для каждой версии движка Godot и называется **шаблоном экспорта (export template)**. Получив шаблон, мы можем экспортировать на

любую из встроенных платформ. Мы можем легко загрузить этот шаблон экспорта из самого редактора Godot, как показано ниже:

1. В верхней строке меню откройте раскрывающийся список **Редактор (Editor)** и нажмите **Управление шаблонами экспорта... (Manage Export Templates...)**.

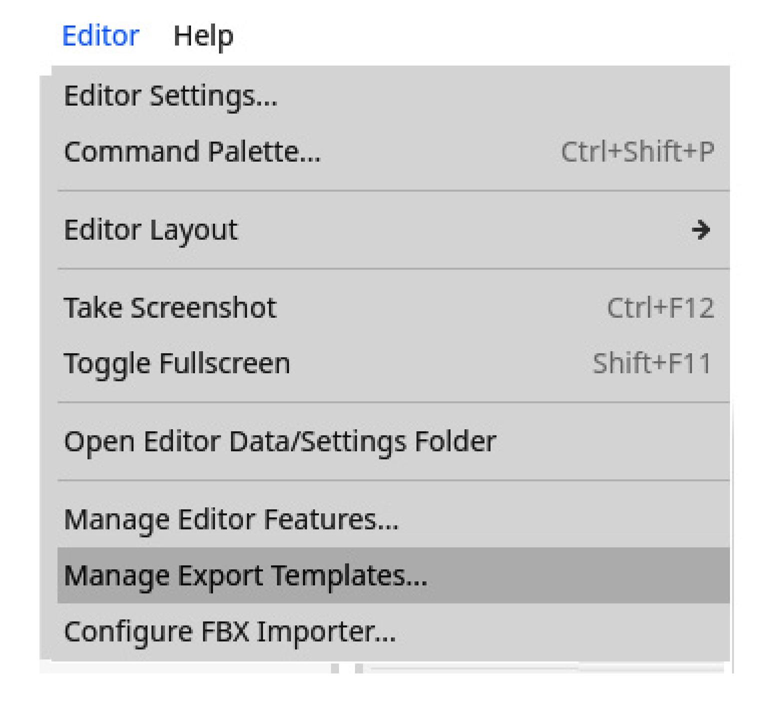


Рисунок 12.1 – Доступ к менеджеру шаблонов экспорта можно получить через раскрывающееся меню «Редактор»

1. Нажмите **Загрузить и установить (Download and Install)**.

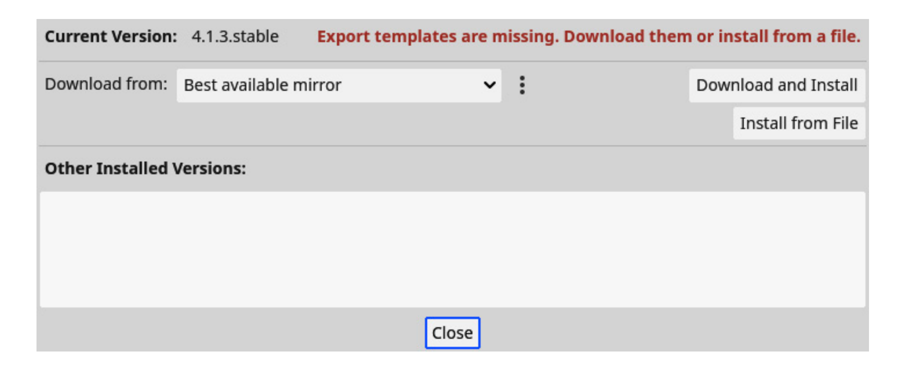


Рисунок 12.2 – В менеджере шаблонов экспорта мы можем загружать и устанавливать шаблоны экспорта

Загрузка и установка шаблона экспорта займет некоторое время, но вам нужно сделать это только один раз для каждой используемой версии Godot Engine. Теперь мы готовы выполнить правильный экспорт.

Осуществление фактического экспорта игры

Имея шаблон экспорта для нашей версии Godot Engine, мы наконец можем экспортировать игру. Мы сделаем это с помощью **Меню экспорта (Export menu)**. Давайте начнём:

1. В верхней строке меню откройте раскрывающийся список **Проект (Project)** и нажмите **Экспортировать... (Export...)**.

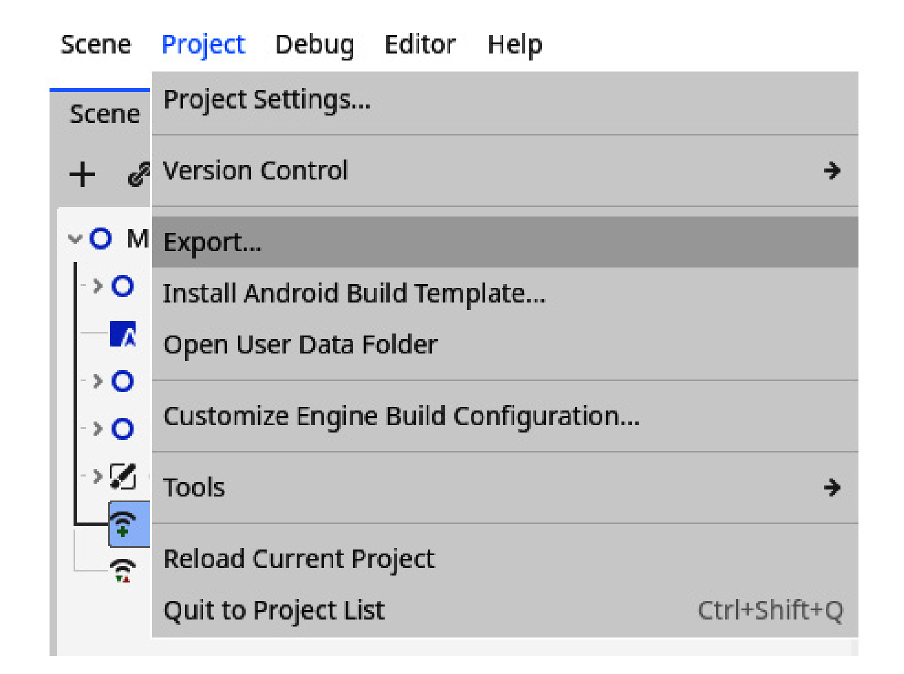


Рисунок 12.3 – Меню «Экспортировать…» доступно через раскрывающееся меню «Проект»

1. Это открывает меню **Export**, которое содержит настройки экспорта для каждого пресета экспорта. В настоящее время оно не содержит никаких пресетов экспорта.
2. Нажмите **Добавить… (Add…)** и выберите компьютерную платформу, которую вы используете в данный момент (**Linux/X11**, **macOS** или **Windows Desktop**).

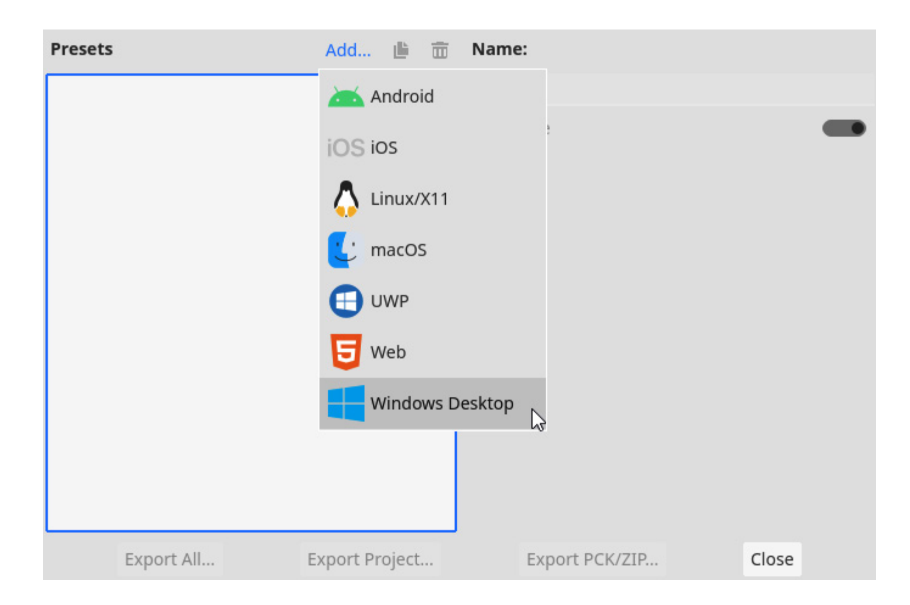


Рисунок 12.4 – Добавление предустановки для определённой платформы в меню «Экспорт»

1. Платформа будет добавлена в список **Предустановки (Presets)**.

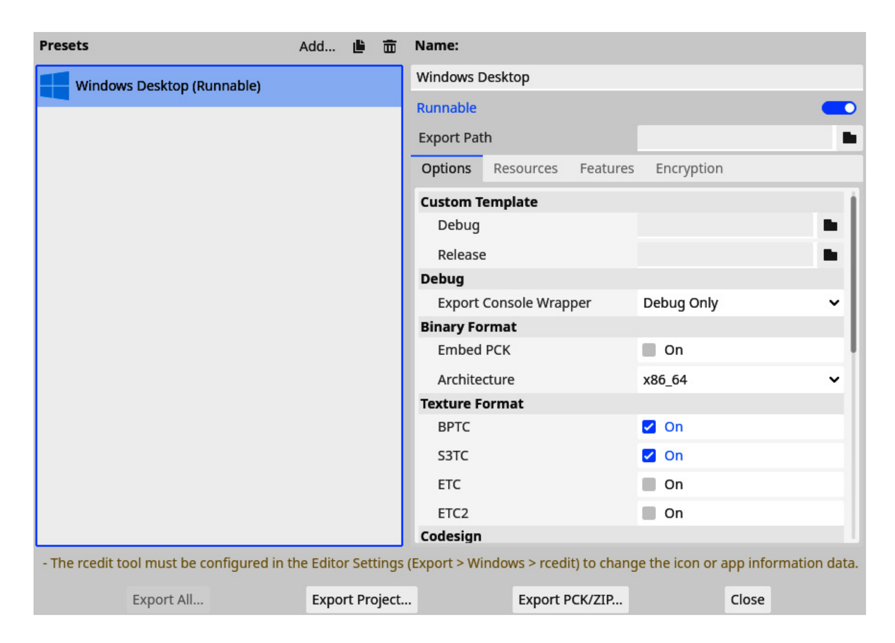


Рисунок 12.5 – Платформа Windows Desktop добавлена как одна из предустановок

1. Настройте экспорт в зависимости от добавленной вами платформы:

    - **Только для macOS**: Перед тем, как экспортировать игру, вам нужно указать идентификатор (bundle identifier) для вашего игрового пакета. Просто заполните это поле **com.survivor.game** или чем-то похожим.

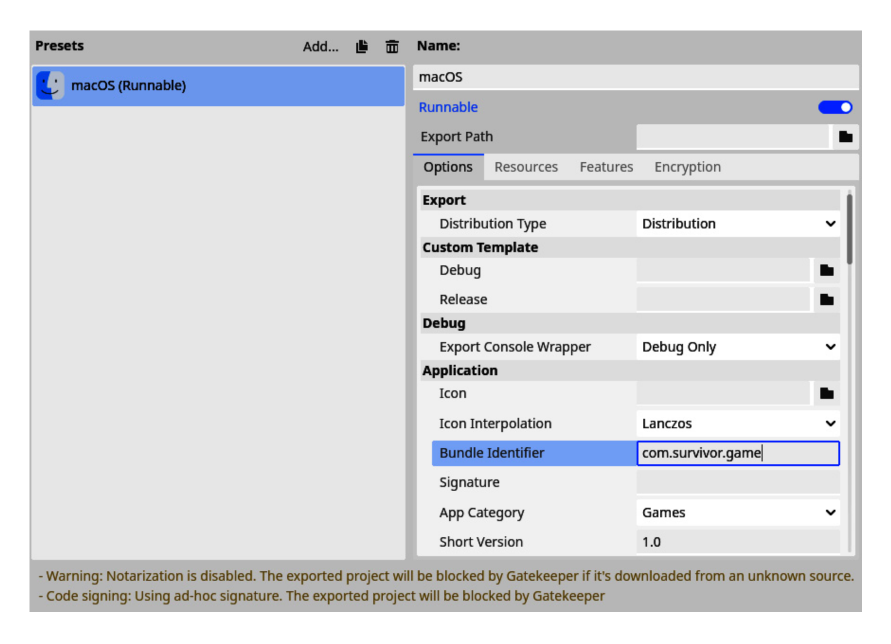


Рисунок 12.6 – Для macOS вам придется указать идентификатор пакета (bundle identifier)

- **Только для Windows и Linux**: включите опцию **Embed PCK**. Это гарантирует, что файл PCK нашей игры (файл пакета, содержащий все данные игры, такие как код и графика) будет встроен в исполняемый файл игры.

Presets Add...
macOS (Runnable)
Windows Desktop (Runna...
Linux/X11 (Runnable)
Name:
Windows Desktop
Runnable
Export Path
Options Resources Features Encryption
Custom Template
Debug
Release
Debug
Export Console Wrapper Debug Only
Binary Format
Embed PCK On
Architecture x86_64
Texture Format
BPTC On
S3TC On
ETC On
ETC2 On
Export All... Export Project... Export PCK/ZIP... Close

Рисунок 12.7 – Для Windows и Linux включите опцию Embed PCK

1. Теперь нажмите **Экспортировать проект... (Export Project...)**, чтобы экспортировать игру.
2. Создайте новую папку с названием **exports**, в которой вы можете создать папку для каждой платформы. Так, если вы экспортируете для Windows, экспортируйте в **exports/windows**. Это просто для того, чтобы придать некоторую структуру нашему экспорту.
3. Отключите опцию **Экспорт с отладкой (Export With Debug)**.

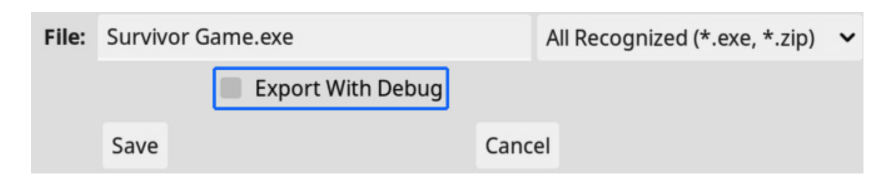


Рисунок 12.8 – Отключение опции «Экспорт с отладкой»

1. Нажмите **Сохранить (Save)**.

Теперь у нас будет экспорт игры для целевой платформы в папке, которую мы только что создали. Повторите шаги в этом разделе для каждой платформы.

Name	Date modified	Type	Size
Survivor Game.exe	07/12/2023 23:24	Application	68.043 KB

Рисунок 12.9 – Экспортированная игра для Windows

Важное примечание

Вы могли заметить, что в меню экспорта есть кнопка **Экспортировать всё... (Export All...)** Когда все платформы, на которые вы хотите экспортировать, настроены, вы можете нажать эту кнопку, чтобы экспортировать их все одновременно.

После экспорта пришло время опубликовать игру и поделиться ею со всем миром.

Загружаем нашу игру на Itch.io

После создания игры очень весело поделиться ею с людьми, которых вы знаете, или с совершенно незнакомыми людьми, чтобы увидеть, как они взаимодействуют и играют с тем, что вы сделали. В этом разделе мы рассмотрим процесс размещения игры на онлайн-платформе Itch.io. Если вы по какой-либо причине не хотите делиться своей игрой, то можете пропустить этот раздел на данный момент; просто вернитесь к нему, когда почувствуете себя готовым. Но не беспокойтесь слишком сильно о том, слишком ли рано делиться игрой или нет. Отзывы игроков всегда хороши, независимо от того, на какой стадии разработки вы находитесь, и выполнение этих шагов по созданию страницы Itch.io также является отличной практикой.

Что такое Itch.io?

Itch.io — это онлайн-платформа и магазин, где люди могут распространять игры, ресурсы (для создания игр) или любые другие цифровые ресурсы, файлы или программы. Он очень известен в сообществе разработчиков игр, потому что многие разработчики размещают там свои небольшие игровые эксперименты, но вы также можете найти более крупные проекты.

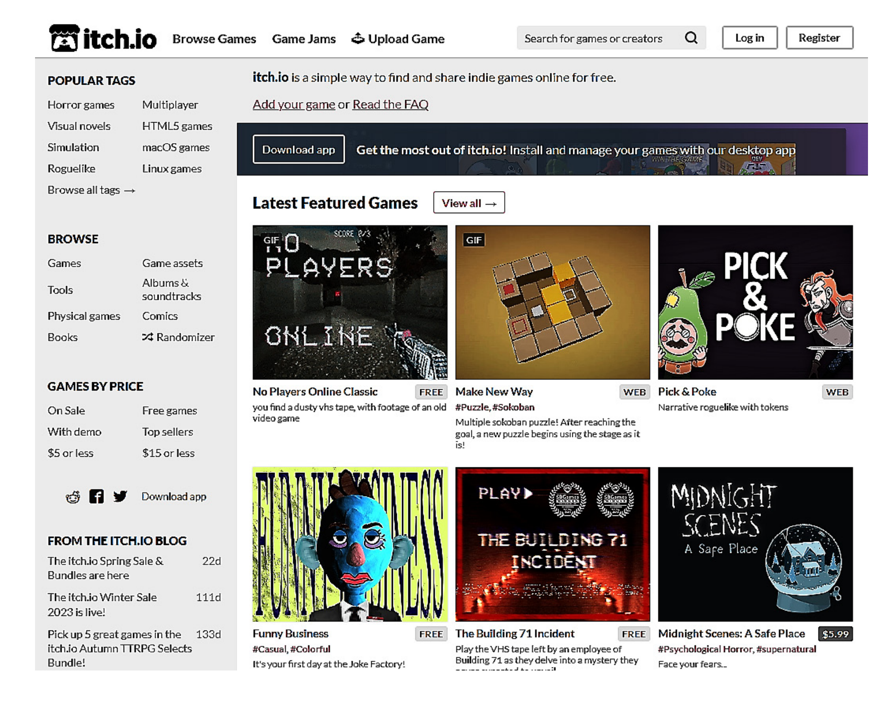


Рисунок 12.10 – Главная страница Itch.io

Itch.io также является местом, где можно поучаствовать в **игровых джемах (game jams)**. Игровой джем — это небольшое онлайн-мероприятие, где вы создаете игру в течение определённого срока на определенную тему. Сроки и тема отличаются от игрового джема к игровому джему. После этого все участники играют в игры друг друга и оставляют отзывы. Это отличный способ попрактиковаться в создании игр и получить отзывы от других разработчиков игр.

Мы загрузим наши компьютерные сборки на платформу Itch.io, но самое крутое, что мы также можем загрузить веб-сборку, которая будет воспроизводиться из браузера, так что людям не придется загружать никаких файлов — они могут просто начать играть. Если мы загрузим и обычный экспорт, и веб-экспорт, люди все равно смогут выбирать, как они хотят играть, что даёт им больше возможностей.

Экспортируем нашу игру для игры в браузере

Веб-экспорт означает, что мы делаем экспорт, который можно включить в любой веб-сайт и воспроизводить в любом браузере, даже мобильном.

Важное примечание

Обратите внимание, что когда мы делаем веб-экспорт, он будет доступен для игры только при запуске через сервер. Вы не сможете напрямую открыть экспорт в своем любимом браузере, не запустив сервер на своем компьютере или не разместив его на онлайн-платформе. К счастью, мы собираемся разместить нашу игру на Itch.io, так что вы сможете играть в неё там.

Первое, что нам нужно сделать, это на данный момент убрать многопользовательский аспект игры.

Удаление мультиплеера

По соображениям безопасности веб-сайт никогда не должен иметь полный доступ к вашему компьютеру. Вот почему веб-экспорт имеет некоторые ограничения, такие как немного худшая производительность и сетевое взаимодействие. Это означает, что многопользовательский режим нашей игры не будет работать в веб-экспорте. Есть способы заставить многопользовательский режим работать для веб-экспорта, но они выходят за рамки этой книги. С одним небольшим изменением он все равно будет нормально работать для одного игрока:

В функцию **_ready()** скрипта **menu.gd** добавьте следующую строку:

```
func _ready():
    _ip_address_line_edit.visible = OS.get_name() != "Web
    # Rest of the _ready function
```

Функция **OS.get_name()**даёт нам имя операционной системы, на которой в данный момент запущена игра. Это гарантирует, что поле ввода IP-адреса больше не будет видно, когда игрок играет через браузер, и не застрянет при попытке подключения, что может сломать игру для него.

Давайте посмотрим, как можно сделать веб-экспорт, отключив многопользовательский режим.

Фактическая реализация веб-экспорта

Процесс веб-экспорта таков же, как и экспорт на компьютерные платформы: мы просто добавляем новый пресет **Web** и экспортируем проект как обычно.

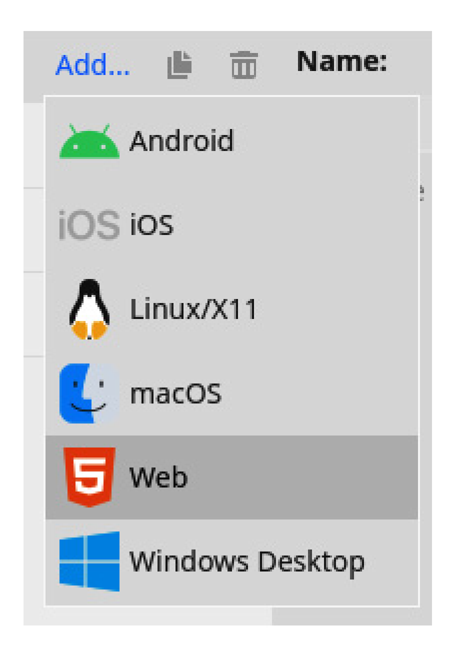


Рисунок 12.11 – Добавление предустановки экспорта для веб-платформы

Единственное, что нам нужно будет сделать, — при сохранении файла назвать его **index.html**, как показано на *Рисунке 12.12*. Itch.io требует это имя, поскольку он будет искать файл **index.html** для запуска игры в браузере.

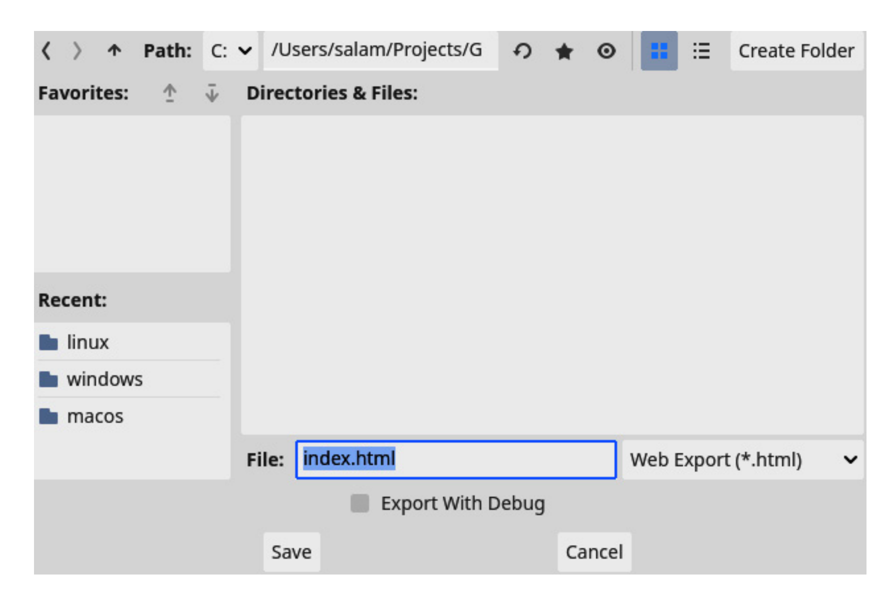


Рисунок 12.12 – Обязательно сохраните экспортируемый файл для веб, как index.html

Теперь, когда у нас есть экспортированная игра, нам нужно сделать из неё zip-файл. Давайте узнаем, как это сделать.

Архивирование веб-экспорта

Itch.io потребуется ZIP-файл, содержащий все экспортированные файлы для веб-экспорта. ZIP-файл — это файл, который объединяет несколько других файлов в сжатый формат, что упрощает их транспортировку, а также уменьшает размер контента.

В зависимости от используемой платформы процесс немного отличается.

Для Windows и macOS выполните следующие действия:

1. Выберите все экспортированные для веб файлы игры.

index.apple-touch-icon.png	07/12/2023 23:24	PNG File	19 KB
index.apple-touch-icon.png.import	07/12/2023 23:09	IMPORT File	1 KB
index.audio.worklet.js	07/12/2023 23:24	JavaScript File	8 KB
index.html	07/12/2023 23:24	Chrome HTML Do...	7 KB
index.icon.png	07/12/2023 23:24	PNG File	4 KB
index.icon.png.import	07/12/2023 23:09	IMPORT File	1 KB
index.js	07/12/2023 23:24	JavaScript File	442 KB
index.pck	07/12/2023 23:24	PCK File	80 KB
index.png	07/12/2023 23:24	PNG File	21 KB
index.png.import	07/12/2023 23:09	IMPORT File	1 KB
index.wasm	07/12/2023 23:24	WASM File	34.581 KB
index.worker.js	07/12/2023 23:24	JavaScript File	6 KB

Рисунок 12.13 – Выбор всех экспортированных файлов

1. Теперь щёлкните по ним правой кнопкой мыши, чтобы появилось меню **Параметры (Options)**.

    - Для **Windows**: в разделе **Отправить(Send to)**, выберите **Сжатая ZIP-папка (Compressed (zipped) folder)**.

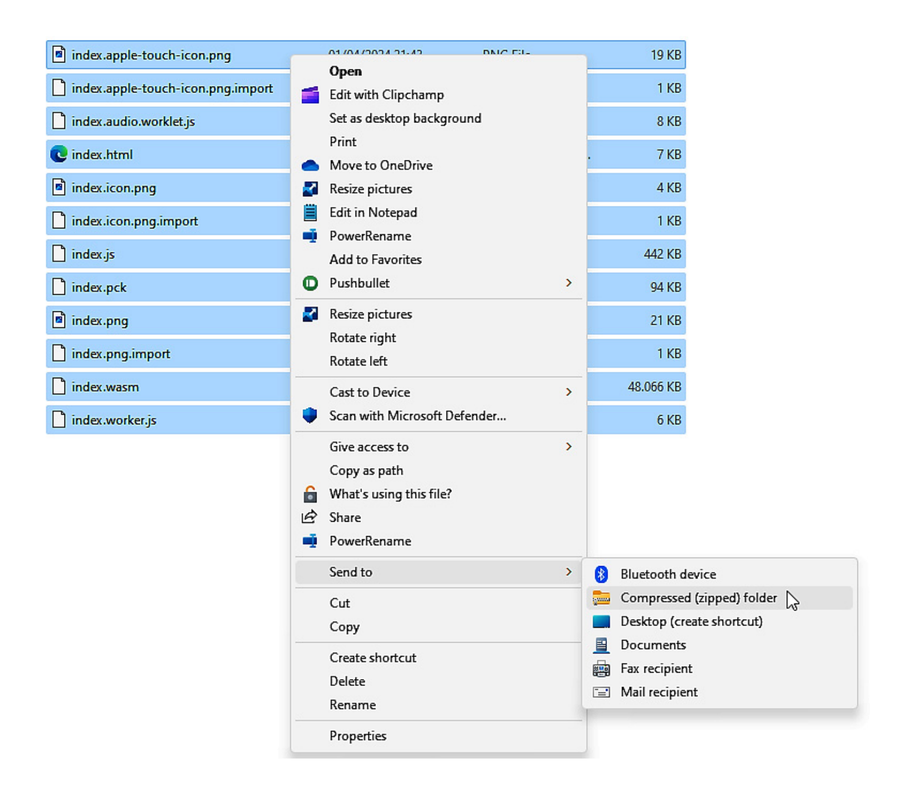


Рисунок 12.14 – На платформе Windows выберите Compressed (zipped) folder

- Для **macOS**: выберите **Сжать (Compress)**.

Рисунок 12.15 – На платформе macOS выберите Compress

1. Теперь назовите ZIP-файл **Survivor Game Web**.

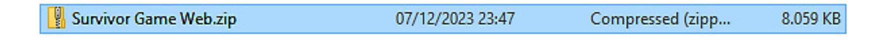


Рисунок 12.16 – Полученный ZIP-файл

Для **Linux** выполните следующие действия:

1. Откройте приложение Terminal на вашем компьютере..
2. Используйте команду **cd ~/path/to/game/export/folder**, указав правильный путь, чтобы перейти к папке веб-экспорта игры.
3. Теперь запустите **zip -r "Survivor Game Web.zip"**, чтобы создать ZIP-файл с содержимым этой папки.

```
 web git:(main) ✗ zip -r "Survivor Game Web.zip" .
adding: index.apple-touch-icon.png (deflated 0%)
adding: index.wasm (stored 0%)
adding: index.icon.png (stored 0%)
adding: index.html (deflated 64%)
adding: index.png (deflated 10%)
adding: index.audio.worklet.js (deflated 70%)
adding: index.apple-touch-icon.png.import (deflated 51%)
adding: index.js (deflated 78%)
adding: index.icon.png.import (deflated 50%)
adding: index.worker.js (deflated 59%)
adding: index.pck (deflated 42%)
adding: index.png.import (deflated 50%)
```

Рисунок 12.17 – Терминал Linux после запуска команды создания ZIP-файла

Когда все экспортированные данные готовы и заархивированы, пришло время загрузить игру на Itch.io.

Загрузка на Itch.io

Теперь, когда все готово, мы можем загрузить нашу игру на платформу Itch.io и создать собственную страницу, на которой люди смогут играть и скачивать игру:

1. Создайте учётную запись на https://itch.io/register. Обязательно отметьте галочкой поле рядом с надписью **Я заинтересован в распространении контента на itch.io (I'm interested in distributing content on itch.io)**, потому что это именно то, что мы хотим сделать.

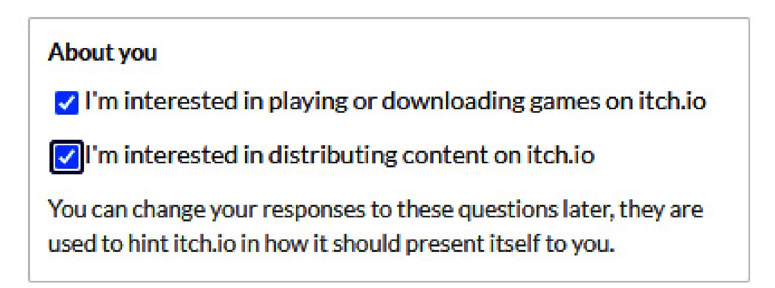


Рисунок 12.18 – При регистрации на Itch.io укажите, что мы

хотим распространять контент

1. После регистрации вам сначала придется подтвердить свой адрес электронной почты, открыв письмо, которое Itch.io отправил на адрес электронной почты, указанный вами для своей учетной записи, и нажав кнопку с надписью **Нажмите, чтобы подтвердить свой адрес электронной почты (Click to verify your email address)**.

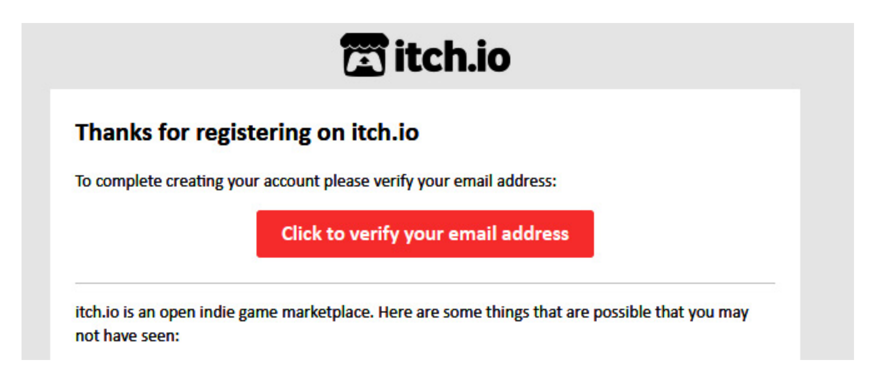


Рисунок 12.19 – Подтвердите свой адрес электронной почты

1. Вернувшись в браузер, вы должны попасть на страницу **Creator Dashboard**. Нажмите большую красную кнопку с надписью **Создать новый проект (Create new project)**.

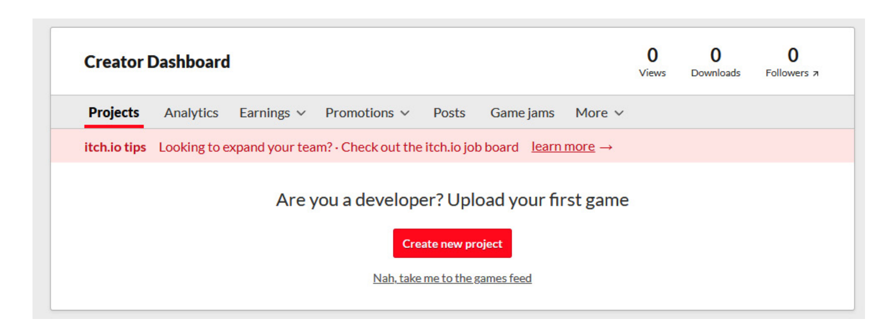


Рисунок 12.20 – Нажмите «Создать новый проект», чтобы начать создание страницы игр

1. Дайте проекту название **Survivor Game**.

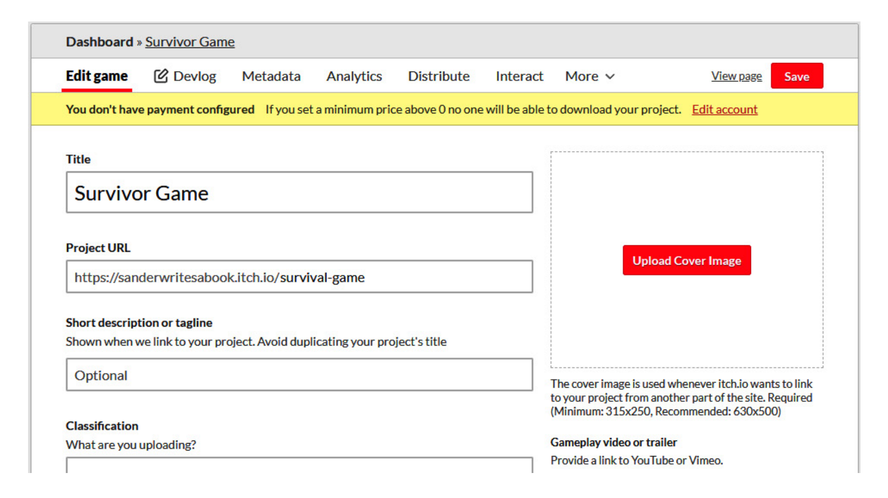


Рисунок 12.21 – Дайте проекту название Survivor Game

1. В поле **Вид проекта (Kind of project)** выберите **HTML**. Это позволит людям играть в игру в своём браузере.

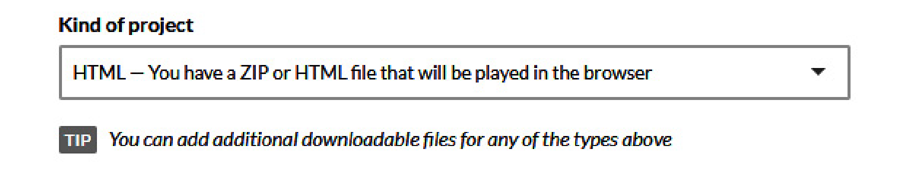


Рисунок 12.22 – Установите тип проекта — HTML

1. Теперь в разделе **Загрузки (Uploads)** загрузите каждый экспортный ZIP-файл.



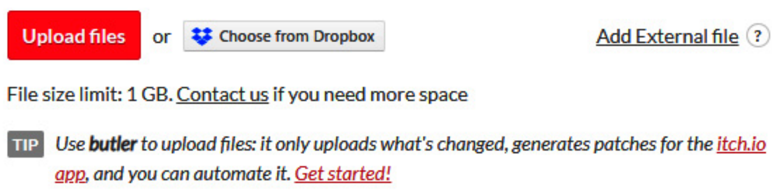


Figure 12.23 – Upload all exports

1. Далее укажите, для какой платформы предназначен

каждый экспортный файл. Это поможет людям понять, какой файл им нужно загрузить для своей платформы.

Рисунок 12.24 – Для платформ Windows, macOS и Linux следует указать платформу

1. Для ZIP-файла содержащего веб-экспорт, выберите **Этот**

файл будет воспроизведен в браузере (This file will be played in the browser).

Survivor Game Web.zip · Success
8mb · Change display name Move
Today at 11:02 PM
More... Delete file
This file will be played in the browser

Рисунок 12.25 – Для ZIP-файла веб-экспорта укажите, что его можно воспроизвести в браузере

1. В разделе **Параметры встраивания (Embed options)** установите **Размер окна просмотра (Viewport dimensions)** на **1152** px × **648** px. Это размер окна нашей игры на странице Itch. Эти размеры являются точными размерами из настроек нашего проекта.

Embed options

How should your project be run in your page?



Viewport dimensions



Рисунок 12.26 — Настраиваем окно показа нашей игры для браузера

1. В разделе **Параметры кадра (Frame options)** включите **Поддержка SharedArrayBuffer (SharedArrayBuffer support)**. Это необходимо для веб-экспорта Godot Engine

4.

Frame options

☐ Mobile friendly — Your project can run on mobile phones (smaller resolution and touch support)

☐ Automatically start on page load — Not recommended for Unity games, since they can lag the browser when loading

☐ Fullscreen button — Add a button to the bottom right corner of your embed to make it fullscreen

☐ Enable scrollbars — Enable scrollbars in the iframe that contains your project

☑ `SharedArrayBuffer` support — **(Experimental)** This may break parts of the page or your project. Only enable if you know you need it. Learn more

Рисунок 12.27 – Включение поддержки SharedArrayBuffer

1. Нажмите **Сохранить и просмотреть страницу (Save & view page)**.

Visibility & access

Use Draft to review your page before making it public. Learn more about access modes

◉ Draft — Only those who can edit the project can view the page

○ Restricted — Only owners & authorized people can view the page

○ Public — Anyone can view the page, **you can enable this after you've saved**

Save & view page

Рисунок 12.28 – Нажмите «Сохранить и просмотреть страницу»

1. Мы перейдём к предварительному просмотру того, как выглядит наша страница. Вы увидите, что игра должна загрузиться в первый раз. Чтобы опубликовать её публично, нам нужно вернуться на страницу **Редактирование (Edit)**.

Loading game for the first time...

More information

Рисунок 12.29 – В первый раз загрузка игры займет немного

больше времени

1. На этот раз выберите **Публичный (Public)** в разделе **Видимость и доступ (Visibility & access)**.

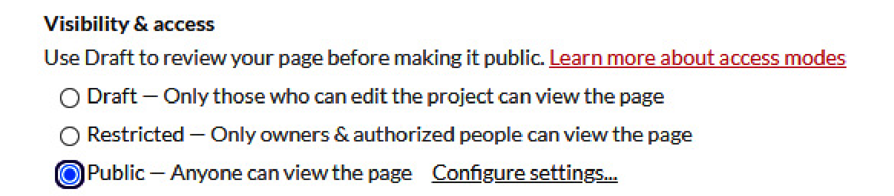


Рисунок 12.30 – Выберите «Публичный» в разделе «Видимость и доступ»

1. Нажмите **Сохранить (Save)** ёще раз.

У игры теперь есть своя страница, и она доступна для всеобщего просмотра! Отправьте ссылку другу и поделитесь ею в социальных сетях — вы опубликовали игру!

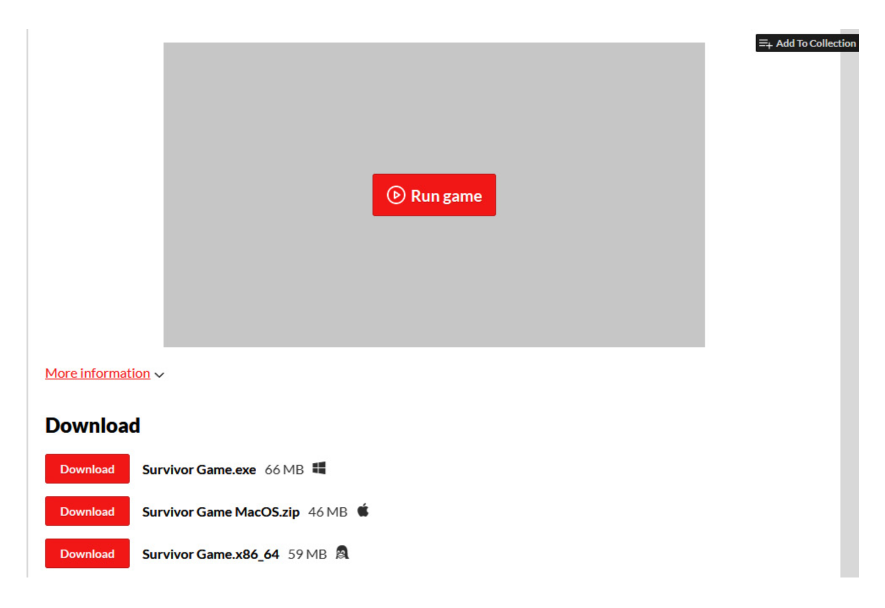


Рисунок 12.31 – Страница нашей игры Itch.io

Нажатие кнопки **Запустить игру (Run game)** запустит нашу игру в браузере, а использование кнопок загрузки в нижней

части страницы поможет нам загрузить игру для любой желаемой платформы.

Мы знаем, как делать базовый экспорт для большинства компьютеров и веба, но что насчет других платформ, таких как мобильные устройства и консоли? Давайте рассмотрим это далее.

Экспорт нашей игры на другие платформы

Теперь, когда наша игра опубликована и в неё может играть каждый, давайте кратко рассмотрим, как экспортировать ее на другие платформы, не являющиеся обычными компьютерами.

Мобильные платформы

Для мобильных устройств, таких как устройства Android и iOS, процесс немного сложнее. Однако, как только процесс будет настроен, он будет очень надёжным. Вы можете найти руководства по экспорту на мобильные платформы в официальной документации Godot Engine:

- Экспорт на Android: https://docs.godotengine.org/en/stable/tutorials/export/exporting_for_android.html
- Экспорт на iOS: https://docs.godotengine.org/en/stable/tutorials/export/exporting_for_ios.html

Помимо простого экспорта игры, вам также придется учитывать тот факт, что мобильные устройства, как правило, не имеют внешних кнопок, поэтому игровой процесс должен предусматривать управление с помощью сенсорного экрана.

Консоли

Давайте рассмотрим «слона в комнате» — как насчёт экспорта на консоли, такие как **PlayStation**, **Xbox** или **Nintendo**?

Ну, хорошая новость в том, что это возможно! Плохая новость в том, что поскольку Godot Engine имеет открытый исходный код, а библиотеки кода, необходимые для экспорта на эти консоли, имеют закрытый исходный код, параметры экспорта для них не могут быть включены в базовую версию Godot. Поэтому по умолчанию их нет в движке.

Однако есть компании, которые предоставляют специализированную версию Godot Engine для экспорта на консоли и/или помогают портировать целую игру. К этим компаниям относятся **W4** (компания, в которой работают многие из разработчиков оригинального Godot Engine), **Pineapple Works** и **Lone Wolf Technology**. Актуальный список можно найти на сайте документации Godot: https://docs.godotengine.org/en/stable/tutorials/platform/consoles.html.

Итоги

В этой главе мы узнали, как экспортировать нашу игру на несколько компьютерных платформ, таких как Windows, Mac и Linux. Мы также увидели, как можно экспортировать для веба и загрузить игру на Itch.io. Теперь мы готовы создать целую игру и опубликовать её!

В следующей главе мы изучим более продвинутые методы объектно-ориентированного программирования.

Опрос

- Что такое шаблоны экспорта?
- Почему мы включили **Embed PCK** при экспорте для Windows и Linux?
- На какие платформы можно экспортировать с помощью Godot Engine?

13

Продолжение ООП и продвинутые темы

В ходе чтения этой книги мы узнали так много о программировании, от базовых переменных, потоков управления и классов до вещей, специфичных для GDScript, таких как доступ к узлам в дереве сцены и специальные аннотации. Однако не заблуждайтесь — есть ещё гораздо больше знаний, которые могут помочь решать проблемы проще и быстрее.

После многих лет обучения и профессионального применения моих навыков программирования я могу с уверенностью заявить, что компьютерные науки — это глубокая и полезная область для дальнейшего изучения. Плюс, каждые несколько лет появляется новая технология, ожидающая изучения.

В этой главе мы рассмотрим ряд более продвинутых приемов и концепций, которые поднимут ваши навыки программирования на новую высоту!

В этой главе мы рассмотрим следующие основные темы:

- Ключевое слово **super**
- Статические переменные и функции
- Перечисления
- Лямбда-функции
- Передача параметров по значению или ссылке
- Аннотация **@tool**

Технические требования

Как и прежде, окончательный код можно найти в репозитории

GitHub в подпапке для этой главы: https://github.com/PacktPublishing/Learning-GDScript-by-Developing-a-Game-with-Godot-4/tree/main/chapter13.

Ключевое слово super

В *Главе 4* мы узнали все о наследовании и о том, как можно переопределять функции из базового класса в унаследованном классе. Это переопределение заменяет функцию совершенно новым телом и отменяет исходную реализацию базового класса. Однако иногда нам все равно хотелось бы выполнить исходную логику, которая была определена в родительском классе.

Для этого мы можем использовать ключевое слово **super**. Это ключевое слово дает нам прямой доступ ко всем функциям родительского класса, на котором основан текущий класс. Рассмотрим следующий пример, где мы хотим иметь в нашей игре разные виды стрел, чтобы стрелять по врагам:

```
class BaseArrow:
   func describe_damage():
      print("Пронзает человека")
class FireArrow extends BaseArrow:
   func describe_damage():
      super()
      print("И поджигает")
```

Здесь мы определяем класс **BaseArrow**, который является базой для всех видов стрел. Он имеет одну функцию **describe_damage()**, которая просто описывает, какой урон наносит стрела, выводя **Пронзает человека** в консоль.

Когда мы переопределяем функцию **describe_damage()** класса **FireArrow**, мы сначала вызываем **super()** как функцию. Это выполнит исходную функцию **describe_damage()** класса **BaseArrow** перед выполнением остальных.

Давайте выполним код, использующий эти классы:

```
var fire_arrow: FireArrow = FireArrow.new()
fire_arrow.describe_damage()
```

Результат будет следующим:

```
Пронзает человека
И поджигает
```

Вы можете видеть, что функция **describe_damage()** из базового класса была выполнена с использованием ключевого слова **super()**, как и остальная часть реализации класса **FireArrow**.

Ключевое слово **super** даёт доступ к базовому классу, от которого мы унаследовались; независимо от того, переопределили ли мы эту функцию или нет, она всегда будет возвращаться к исходной. Давайте продолжим с рассмотрением еще одного нового ключевого слова – **static**.

Статические переменные и функции

Следующее ключевое слово, которое мы рассмотрим, — это **static**. . Мы можем объявить переменную или функцию как статическую, поместив это ключевое слово перед ней:

```
class Enemy:
   static var damage: float = 10.0
   static func do_battle_cry():
      print("Рррррррры!")
```

Статические переменные и функции объявляются в самом классе. Это означает, что к ним можно получить доступ без создания экземпляра класса:

```
print(Enemy.damage)
```

```
Enemy.do_battle_cry ()
```

Статические переменные созданы для хранения информации, которая привязана к полному классу объектов. Но будьте осторожны — вот два больших подвоха для статических переменных и функций:

- В GDScript статическим переменным можно присвоить новое значение, и вы можете изменять их во время выполнения игры. В идеале этого делать не следует, поскольку это может повлиять на вашу программу способами, которые будет трудно отладить.
- Из статической функции можно вызывать другие функции и использовать переменные-члены класса, но только если они тоже определены как статические. Поскольку статические функции определяются в самом классе, они не имеют всего контекста инициализированного объекта этого класса. Статические функции должны быть очень самодостаточными.

В общем, вы не так часто увидите статические переменные и функции в GDScript, но это известная концепция, которую используют многие объектно-ориентированные языки программирования, такие как C+ + или Java. Давайте рассмотрим перечисления далее.

Перечисления

Enums, сокращение от **перечисления (enumerations)**, — это тип переменной, определяющий набор констант, которые необходимо сгруппировать. В отличие от обычных констант, где мы хотим хранить определённое значение, enums автоматически присваивают значения константе.

В *Главах 2* и *5* мы увидели, что очень важно иметь хорошо именованные переменные. Таким образом, мы всегда знаем, что они будут содержать. На самом деле мы можем сделать это и для значений переменных, с именованными значениями. Используя именованные значения, мы можем связать понятное

человеку имя с определённым значением, делая код более читаемым. Это также удаляет магические числа из кода. Взгляните на это перечисление:

```
enum DAMAGE_TYPES {
   NONE,
   FIRE,
   ICE
}
```

Здесь мы создаём перечисление с именем **DAMAGE_TYPES**, которое определяет три именованных значения – **NONE**, **FIRE** и **ICE**. Вы можете получить доступ к этим значениям следующим образом:

```
DAMAGE_TYPES.FIRE
```

Давайте попробуем их распечатать:

```
print(DAMAGE_TYPES.NONE)
print(DAMAGE_TYPES.FIRE)
print(DAMAGE_TYPES.ICE)
```

Вы увидите, что он выведет следующее:

```
0
1
2
```

Это потому, что каждое из имён в перечислении связано с целочисленным значением. Однако вместо использования этих грубых целых чисел мы теперь можем использовать в коде хорошо читаемые имена. Первое именованное значение связано с **0**, а каждое последующее увеличивается на единицу.

Перечисление также можно использовать для указания типа переменных; таким образом, мы узнаем, что переменной

необходимо присвоить значение перечисления определённого типа:

```
var damage_type: DAMAGE_TYPES = DAMAGE_TYPES.FIRE
match damage_type:
   DAMAGE_TYPES.NONE:
      print("Ничего особенного не происходит")
   DAMAGE_TYPES.FIRE:
      print("Вы загорелись!")
   DAMAGE_TYPES.ICE:
      print("Вы замерзаете!")
```

В этом примере мы указываем тип переменной **damage_type** как **DAMAGE_TYPES**. Затем мы можем, например, сопоставить (match) эту переменную и определить, что делать.

Перечисления против строк

Теперь вы можете подумать: — «*Почему бы нам не использовать строки, если мы хотим иметь возможность прочитать значение*»? Ну, просто потому, что строки медленнее и требуют больше памяти для работы, чем целые числа — базовый тип данных значений перечисления. Другая причина — простота использования. Перечисление имеет конечный набор значений, тех, которые мы определили, в то время как строка может иметь произвольное количество символов. Поэтому при использовании перечисления мы уверены, что имеем дело только с известными нам значениями.

Мы также можем получить доступ к перечислениям, которые определены в одном классе, из совершенно другого класса; как и к статическим переменным и функциям, к ним можно получить доступ напрямую из типа класса:

```
class Arrow:
   Enum DAMAGE_TYPES {
      NONE,
      FIRE
   }
```

```
func _ready():
   var damage_type: Arrow.DAMAGE_TYPES = Arrow.DAMAGE_TY
```

Здесь мы определяем перечисление **DAMAGE_TYPES** в классе **Arrow** . Позже мы можем получить доступ к этому перечислению, используя **Arrow.DAMAGE_TYPES** напрямую.

В этом разделе мы рассмотрели перечисления, именованные значения, которые помогают нам, предоставляя понятные человеку метки. Далее мы рассмотрим лямбда-функции.

Лямбда-функции

До сих пор каждая функция, которую мы написали, принадлежала классу или файлу, который можно было бы рассматривать как класс, но на самом деле есть способ определить функции отдельно от любого определения класса. Такие функции называются **лямбда-функции (lambda functions)**.

Создание лямбда-функции

Давайте рассмотрим лямбда-функцию:

```
var print_hello: Callable = func(): print("Привет!")
```

Вы видите, что мы определили функцию, как мы это обычно делаем, но на этот раз без имени функции. Вместо этого мы присвоили функцию переменной. Эта переменная теперь содержит функцию в форме типа объекта **Callable**. Мы можем вызвать объект **Callable** позже, например так:

```
print_hello.call()
```

Это запустит определённую нами функцию и выведет **Привет!** в консоль.

Лямбда-функции, как и обычные функции, также могут принимать аргументы:

```
var print_largest: Callable = func(number_a: float, numb
   if number_a > number_b:
      print(number_a)
   else:
      print(number_b)
```

В этом примере вы также можете видеть, что лямбда-функции могут содержать несколько строк кода в виде блока кода, где каждая строка имеет одинаковый уровень отступа.

Где используются лямбда-функции

Итак, где мы будем использовать лямбда-функции? Что ж, они очень полезны в скриптах, где вам нужна относительно небольшая функция, но вы не хотите, чтобы она постоянно находилась в классе.

Одним из замечательных применений лямбда-функций является подключение сигналов. Если у нас есть кнопка, например, то мы можем подключиться к её сигналу нажатия, используя лямбда-функцию, следующим образом:

```
button.connect("pressed", func(): print("Кнопка нажата!"
```

Теперь каждый раз, когда кнопка нажимается и испускает сигнал **pressed** , наша лямбда-функция выполняется и выводит сообщение **Кнопка нажата!**.

Ещё одним вариантом его использования являются функции **filter()** или **sort_custome()**, которые могут использовать лямбда-функцию для фильтрации или сортировки элементов в массиве:

```
[0, 1, 2, 3, 4].filter(func(number: int): return number 
[0, 3, 2, 4, 1].sort_custome(func(number_a: int, number_
```

Каждый массив имеет функции **filter()** и **sort_custome()**, которые принимают **Callable** в качестве аргумента. Функция **filter()** отфильтрует любой элемент в массиве, для которого функция возвращает **false**, в результате чего получится массив, содержащий только те элементы, для которых функция возвращает **true**. В предыдущем примере это приводит к массиву, содержащему только чётные числа.

Функция **sort_custome()** сортирует элементы в массиве, используя предоставленный нами **Callable** Лямбда-функция должна принимать два элемента и, когда первый элемент должен быть отсортирован до второго, возвращать **true**; в противном случае она должна возвращать **false**. Таким образом, мы можем определить собственные правила сортировки элементов массива.

Результирующие массивы после запуска **filter()** и **sort_custome()** с нашими лямбда-функциями выглядят следующим образом:

```
[0, 2, 4]
[0, 1, 2, 3, 4]
```

Дополнительная информация

Дополнительную информацию о лямбда-функциях можно найти в официальной документации: https://docs.godotengine.org/en/stable/classes/class_callable.html.

Теперь, когда мы знаем, что такое лямбда-функции, давайте рассмотрим различные способы передачи значений функциям.

Передача параметров по значению или ссылке

При передаче параметров в функцию на самом деле есть два разных способа, которыми эти параметры могут попасть в тело этой функции — по значению или по ссылке. Мы, как

программисты, не выбираем, какой из двух вариантов использовать; GDScript принимает это решение на основе типа данных значения, которое мы предоставляем функции. Давайте подробнее рассмотрим оба метода передачи значений, какие типы данных применяются к каждому из них и почему важно знать разницу.

Передача по значению

Передача по значению (passing by value) означает, что GDScript отправляет точную копию значения в функцию. Этот подход очень прост и предсказуем, поскольку мы получаем новую переменную в вызываемой функции. Однако, поскольку копирование данных занимает время, оно может быть довольно медленным для больших типов данных.

Типы данных, которые передаются по значению, — это любые простые встроенные типы данных, такие как целые числа, числа с плавающей точкой и логические значения. Некоторые немного более сложные классы, такие как строки, **Vector2** и **Colors** также передаются по значению. Этот список не является исчерпывающим. Общее правило включения — всё, что не является массивом, не является словарем и не унаследовано от класса **Object**, передаётся по значению.

Давайте посмотрим, как передача по значению выглядит на практике:

```
func _ready():
   var number: int = 5
   print("Число перед функцией: ", number)
   function_taking_integers(number)
   print("Число после функции: ", number)
   var string: String = "Привет!"
   print("Строка перед функцией: ", string)
   function_taking_strings(string)
   print("Строка после функции: ", string)
func function_taking_integers(number: int):
   number += 10
```

```
    print("Число во время функции: ", number)
func function_taking_strings(string: String):
    string[0] = "К"
    print("Строка во время функции: ", string)
```

Здесь вы можете видеть, что у нас есть две функции, которые принимают целое число и строку соответственно, и каждая из них изменяет значение параметра во время своего выполнения. Мы также выводим значение этого целого числа и строки на каждом шагу, до, во время и после выполнения функции, чтобы увидеть, была ли изменена исходная переменная из функции **_ready()**. Запуск этого выведет следующее:

```
Число перед функцией: 5
Число во время функции: 15
Число после функции: 5
Строка перед функцией: Привет!
Строка во время функции: Кривет!
Строка после функции: Привет!
```

Мы видим, что, хотя значение и изменилось каким-то образом во время выполнения функции, исходное значение не изменилось. В этом и заключается веселье передачи по значению; нам не нужно беспокоиться о побочных эффектах.

Побочные эффекты функций

Побочные эффекты (side effects) на языке программистов означают, что функция изменяет состояние программы способами, которые неочевидны, изменяя переменные за пределами её области действия. Вы хотите избегать этого, насколько это возможно, чтобы было легко понять, что делает функция.

Вот как работает передача по значению – просто прямая копия данных. Теперь рассмотрим противоположную идею – передачу по ссылке.

Передача по ссылке

Другой способ передачи значений в функцию — по ссылке (reference). Это означает, что GDScript не копирует всё значение, а отправляет ссылку, которая указывает на значение. Эта ссылка указывает на место, где хранится фактическое значение, и может использоваться для доступа к нему и его изменения.

Этот режим передачи параметров используется для массивов, словарей и любого класса, который наследует от класса **Object**, который включает все типы узлов. По сути, он используется для передачи больших типов данных, поскольку копирование их полного значения заняло бы слишком много времени и замедлило бы выполнение нашей игры.

Вот пример того, как выглядит передача по ссылке:

```
func _ready():
   var dictionary: Dictionary = { "value": 5 }
   print("Словарь перед функцией: ", dictionary)
   function_taking_dictionary(dictionary)
   print("Словарь после функции: ", dictionary)
func function_taking_dictionary(dictionary: Dictionary):
   dictionary["a_value"] = "изменилось"
   print("Словарь во время функции: ", dictionary)
```

Вновь мы используем ту же настройку, что и раньше, чтобы распечатать наш словарь на каждом этапе пути. Мы запускаем код и получаем следующий вывод:

```
Словарь перед функцией: { "value": 5 }
Словарь во время функции: { "value": 5, "a_value": "изме
Словарь после функции: { "value": 5, "a_value": "изменил
```

Как и ожидалось, мы видим, что после выполнения функции исходный словарь из функции **_ready()** также был изменён! Это побочный эффект в действии.

В целом, хорошей практикой является никогда не изменять значения и переменные, которые входят в функцию, и всегда

делать копию или использовать их напрямую для вычисления значения для другой переменной. Если есть сомнения, лучше всего проверить, передается ли значение по значению или по ссылке; таким образом вы никогда не столкнётесь с непреднамеренными ошибками.

Дублирование массивов и словарей

Если вы действительно хотите сделать копию массива или словаря, то вы можете использовать функцию **duplicate()**, которая определена для этих типов данных. Эта функция вернёт копию массива или словаря, которую вы можете безопасно изменять.

Более подробную информацию смотрите в документации: https://docs.godotengine.org/fr/4.x/classes/class_array.html#class-array-method-duplicate.

Теперь перейдем к рассмотрению того, как можно создавать инструменты для редактора Godot непосредственно из него самого.

Аннотация @tool

Помимо использования GDScript для запуска кода во время выполнения нашей игры, мы можем использовать его для запуска кода в самом редакторе. Запуск кода в редакторе дает нам возможность визуализировать такие вещи, как высота прыжка персонажа, или автоматизировать наш рабочий процесс. Поступая так, мы расширяем редактор Godot для наших собственных конкретных нужд. Существует несколько способов запуска кода GDScript в редакторе, от запуска отдельных скриптов до написания целых плагинов, но самый простой способ — использовать аннотацию **@tool**.

Аннотация **@tool** — это аннотация, которую можно добавить в начало любого скрипта. Ее эффект заключается в том, что узлы с этим скриптом будут запускать свой скрипт в редакторе, как если бы они были созданы в игре. Это означает, что весь их код запускается из редактора.

Это очень полезно, когда мы редактируем наши сцены и хотим предварительно просмотреть некоторые вещи в редакторе, например, состояние нашего игрока, или создать новые узлы с помощью кода.

Зная это, мы можем скорректировать наш скрипт проигрывателя, добавив аннотацию **@tool** вверху, чтобы обновить метку здоровья в редакторе:

```
@tool
class_name Player extends CharacterBody2D
const MAX_HEALTH: int = 10
@onready var _health_label: Label = $Health
@export var health: int = 10:
   set(new_value):
      health = new_value
      update_health_label()
func _ready():
   update_health_label()
func update_health_label():
   if not is_instance_valid(_health_label):
      return
   _health_label.text = str(health) + "/" + str(MAX_HEAL
```

Этот пример — минимальный объем кода, необходимый для обновления метки здоровья из редактора. Однако вы можете просто добавить аннотацию **@tool** в начало вашего существующего скрипта игрока, и это сработает. Вы увидите, что каждый раз, когда вы теперь меняете здоровье игрока из редактора, метка здоровья будет автоматически отражать это изменение.

Риски @tool

Аннотация **@tool** очень мощная, но не без опасности. Она может навсегда удалить что-то из сцены и легко изменить значения узлов, если вы не будете осторожны, поэтому относитесь к ней с осторожностью.

Однако иногда вы хотите использовать узел в игре и иметь

некоторый код, который выполняется в редакторе. Когда мы это делаем, нам нужен способ отличить, выполняется ли код в игре или редакторе. Это можно сделать с помощью **Engine.is_editor_hint()**. Эта функция глобального объекта **Engine** возвращает **true**, если мы запускаем код из редактора, и **false**, если из игры:

```
if Engine.is_editor_hint():
   # Код для выполнения в редакторе.
if not Engine.is_editor_hint():
   # Код для выполнения в игре.
```

Этот пример кода показывает нам, насколько легко отличить выполнение кода в редакторе от выполнения кода в игре.

Дополнительная информация

Хотите узнать больше об аннотации **@tool** и запуске кода в редакторе? Ознакомьтесь с официальной документацией: https://docs.godotengine.org/en/stable/tutorials/plugins/running_code_in_the_editor.html.

Используя аннотацию **@tool** с умом, мы можем сделать наш рабочий процесс проще и быстрее. Возможности безграничны; вы даже можете получить доступ и изменить почти каждый аспект редактора Godot из одного из этих скриптов, но это выходит за рамки этой книги.

Итоги

В этой главе мы глубже погрузились в некоторые из более сложных тем программирования с помощью GDScript. Мы расширили свои знания объектно-ориентированного программирования с помощью ключевых слов **super** и **static** и разницы между передачей по значению или ссылке. Затем мы увидели больше возможностей языка программирования GDScript, таких как перечисления и лямбда-функции. Мы завершили главу способом запуска кода в самом редакторе Godot, используя аннотацию **@tool**.

Опрос

- Представьте, что у нас есть класс с именем **Character**, в котором есть функция **move()**. Теперь мы создаём класс **Player**, который наследует от этого класса **Character** и переопределяет эту функцию **move()**. Но вместо того, чтобы полностью переопределять её, мы хотим расширить исходную функциональность функции **move() класса Character** . Какое ключевое слово мы можем использовать для вызова исходной функции **move()** класса **Character** из класса **Player**?
- Могут ли функции, помеченные как **static**, вызывать функции, не помеченные как **static**?
- Что выведет на печать следующий фрагмент кода?

```
enum COLLECTIBLE_TYPE {
   HEALTH,
   UPGRADE,
   DAMAGE,
}
print(COLLECTIBLE_TYPES.DAMAGE)
```

- Являются ли типы контейнеров, такие как массивы и словари, единственными типами, которые передаются по ссылке?
- Какую аннотацию мы используем в верхней части скрипта, если хотим запустить его в редакторе?

14

Расширенные шаблоны программирования

Хотя компьютерная наука как научная область довольно новая, ей менее 80 лет, многие умные люди ее изучали. Это означает, что большинство проблем программирования уже встречались в той или иной форме. Это такие проблемы, как, например, как соединить части программы без их жесткой привязки или как создать и уничтожить тысячи объектов, например, пуль, не замедляя игру.

Эти умные люди, которых часто называют архитекторами программного обеспечения, придумали умные решения, которые решают эти проблемы элегантным образом. Затем они поняли, что могут обобщить эти решения в своего рода рецепт, шаблон, который другие тоже могли бы использовать. Это то, что мы называем шаблоном программирования. В этой главе мы узнаем, что такое шаблоны программирования, и рассмотрим три наиболее используемых шаблона в разработке игр.

Чем больше ваш словарный запас шаблонов программирования, тем легче вам будет решать собственные проблемы и доносить свои идеи до других.

В этой главе мы рассмотрим следующие основные темы:

- Основы шаблонов программирования
- Шина событий
- Пул объектов
- Машина состояний

Технические требования

Как и прежде, окончательный код можно найти в репозитории GitHub в подпапке для этой главы: https://github.com/PacktPublishing/Learning-GDScript-by-Developing-a-Game-with-Godot-4/tree/main/chapter14.

Код, необходимый для реализации объектного пула в нашей игре, можно найти здесь: https://github.com/PacktPublishing/Learning-GDScript-by-Developing-a-Game-with-Godot-4/tree/main/chapter14-objectpool.

Что такое шаблоны программирования?

Буду честен – мы не первые, кто создаёт игры или пишет программное обеспечение, если уж на то пошло. Но это на самом деле хорошо; это означает, что многие другие до нас сталкивались с теми же проблемами, которые могли быть и у нас. Они придумали решения этих проблем тем или иным способом, и теперь мы можем использовать эти решения в наших собственных играх и программном обеспечении.

Шаблоны программирования (Programming patterns) или шаблоны проектирования программного обеспечения (software design patterns) — это описания или шаблоны, которые говорят нам, как мы можем решить определённые проблемы во время программирования. Они не являются полностью реализованными решениями; они просто дают нам указания о том, как мы можем справиться с тем, что мы пытаемся решить. Шаблон программирования говорит нам, как мы можем организовать наш код для различных результатов.

Для выражения этих моделей важны различные части:

- **Название**: как называется шаблон.
- **Проблема**: какую задачу пытается решить шаблон.
- **Решение**: как будет работать шаблон.

Шаблоны программирования не только предоставляют решение, но и дают нам возможность рассказать о нашем

программном обеспечении. Рассказать о том, что мы сделали, может быть непросто, поскольку, как и в случае с разработкой программного обеспечения, к каждой проблеме можно подойти множеством различных способов. Шаблоны проектирования дают нам возможность говорить о решении, не вдаваясь слишком глубоко в фактическую реализацию.

Наконец, если мы структурируем наш код в соответствии с одним или несколькими шаблонами, мы знаем, чего ожидать. Мы знаем, как код будет реагировать на новые изменения и взаимодействовать с другими частями программы. Это помогает нам понять, как мы можем вносить изменения, будь то устранение ошибки, добавление новой функциональности или переписывание старого кода.

Переписывание старого кода

Иногда вы обнаружите, что способ, которым вы решили проблему, не является быстрым, расширяемым или достаточным. В этот момент вы можете решить переписать часть кода. Мы называем это рефакторингом кода.

Хотите верьте, хотите нет, но шаблоном может быть что угодно, и вы могли использовать существующий шаблон, даже если не осознавали этого.

Однако это не так просто, как выбрать любой шаблон и вставить его в код нашей игры. Мы должны тщательно обдумать, использовать ли тот или иной шаблон. Когда мы используем определенный шаблон, который плохо подходит для проблемы, которую мы пытаемся решить, и фактически ухудшает наше программное обеспечение, мы называем это **анти-паттерн**.

Чтобы не перегружать эту главу, примеры кода будут скорее служить введением и демонстрацией того, как можно использовать шаблон. Мы не будем реализовывать все шаблоны в нашей игре, так как для некоторых это потребовало бы слишком много текста. Настоящая цель этой главы — познакомить вас с наиболее полезными шаблонами программирования.

Исследуем шину событий

Первый шаблон программирования, который мы рассмотрим, — это **Шина событий (Event Bus)**. Он поможет нам разделить код, то есть двум частям кода не придётся слишком сильно полагаться друг на друга, при этом они всё ещё смогут общаться.

Давайте рассмотрим, какую проблему пытается решить шаблон **Шина событий (Event Bus)**.

Проблема

Если мы разделим разные классы и части нашего кода, их будет проще повторно использовать в дальнейшем. Мы делали это ранее, например, в *Главе 9*, с сигналами, которые мы можем соединить, предоставляемыми Godot Engine. Часть кода, которая публикует сигнал, не заботится о том, кто слушает или хочет получить этот сигнал.

Но сигналы работают только локально, и их глобальное использование может оказаться сложной задачей. Классический пример — система достижений. Достижения — это небольшие награды, которые игрок получает за выполнение определённых задач в игре. Они даже могут быть связаны с внешними системами достижений, такими как в Steam или PlayStation Network. Задачи, необходимые для разблокировки этих достижений, часто привязаны к совершенно разным системам в игре — «*победить финального босса*», «*прыгнуть 250 раз*», «*играть в обратном порядке в течение 2 минут*» и так далее. Из-за этого различия в различных достижениях система достижений должна получать информацию из множества разных частей кода. Однако мы не хотим получать доступ к системе достижений напрямую из кода каждой системы или наоборот, так как это создало бы жесткую зависимость для системы достижений, которая присутствовала бы все время. Например, на Nintendo Switch нет системы достижений, поэтому весь этот код достижений был бы бесполезен.

Теперь, когда мы знаем, какую проблему мы пытаемся решить, давайте углубимся в решение, которым является шина событий (Event Bus).

Решение

Вот тут-то и появляется шаблон **Шина событий**. Это класс, который мы автоматически загружаем. В нём другие фрагменты кода могут подписываться на события или публиковать их. Базовая структура выглядит так:

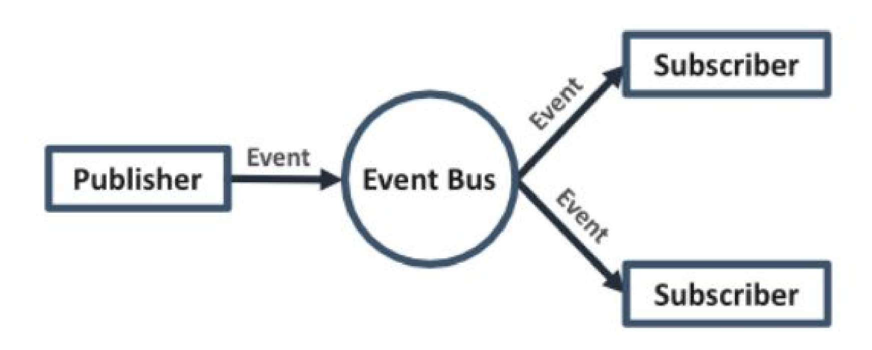


Рисунок 14.1 – Базовая структура работы шины событий

Это очень похоже на сигналы, но на этот раз в глобальном масштабе. Давайте рассмотрим очень простой пример:

```
extends Node
var _observers: Dictionary = {}
func subscribe(event_name: String, callback: Callable):
   if not event_name in _observers:
      _observers[event_name] = []
   _observers[event_name].append(callback)
func publish(event_name: String):
   if not event_name in _observers:
      return
   for callable: Callable in _observers[event_name]:
      callable.call()
```

Добавьте этот скрипт в автозагрузку проекта, как мы это

делали для менеджера рекордов в *Главе 10*, чтобы иметь к нему глобальный доступ.

Давайте рассмотрим очень простой пример битвы с боссом, где у нас есть один узел со скриптом, который выглядит следующим образом:

```
extends Node
func _ready():
   while randf() < 0.99:
      print("Ты всё ещё сражаешься с боссом!")
   print("Босс умирает x.x")
   EventBus.publish("killed_boss")
```

Вы можете видеть, что мы просто сравниваем случайное значение, пока оно не станет больше 0.99, на это должно быть потрачено примерно около 100 попыток. В конечном итоге битва с боссом заканчивается, и мы публикуем в шину событий (Event Bus) событие, которое называется **killed_boss**.

Теперь мы можем создать небольшую систему достижений и подписаться на это событие, чтобы получать уведомления об окончании битвы с боссом:

```
extends Node
func _ready():
   EventBus.subscribe("killed_boss", on_boss_killed)
func on_boss_killed():
   print("Достижение разблокировано: Убейте босса")
```

Добавьте этот скрипт как автозагрузку, назовите его **AchievementSystem**, и тогда вы сможете запустить проект. В консоли вы увидите, что он отлично работает:

```
Вы сражаетесь с боссом!
...
Вы сражаетесь с боссом!
Босс умирает x.x
```

```
Достижение разблокировано: Убейте босса
```

Сигналы, которые являются стандартными в Godot, и шаблоны шины событий являются близкими родственниками шаблона **Наблюдатель (Observer pattern)**. Большое различие между сигналами и шиной событий заключается в том, что с сигналами вы можете подписаться только на одну конкретную сущность, например, когда мы подписываемся на сигнал **died** одного врага, чтобы отметить, что враг умер, тогда как шина событий является глобальной. Неважно, какой узел или объект выдал событие. Все, кто подписан на событие, получат уведомление, например, когда любой узел (неважно какой) выдаёт событие **game_over**, чтобы обозначить, что игра закончилась. Шаблон наблюдателя и все его различные формы широко известны.

Узнать больше

Подробнее о шаблоне Наблюдатель (Observer) можно узнать здесь: https://gameprogrammingpatterns.com/observer.html.

Шаблон программирования **Шина событий (Event Bus)** идеально подходит для развязки вашего кода. Давайте теперь рассмотрим шаблон, который имеет совершенно иное назначение, а именно оптимизацию времени загрузки.

Понимание пула объектов

Второй шаблон программирования, который мы увидим, — это пул объектов (Object Pooling). Цель этого шаблона — поддерживать частоту кадров нашей игры, сохраняя при этом возможность создавать и уничтожать множество объектов или узлов. Давайте глубже погрузимся в то, что мы пытаемся решить, то есть в проблему.

Проблема

В некоторых играх мы хотим иметь возможность создавать и удалять объекты очень быстро. Например, в небольшой игре,

которую мы создали в ходе обучения, мы хотим иметь возможность создавать и удалять снаряды и стрелы быстро и надёжно. При той скорости, с которой сейчас выпускаются наши стрелы, это небольшая проблема, но она может стать таковой, если мы увеличим эту скорость, особенно в многопользовательском режиме. Создание новых узлов — например, с помощью функции **instantiate()**, которую мы видели в *Главе 10*, и добавление их в дерево сцены происходит довольно медленно. Игре необходимо загружать файл сцены с диска, а затем выделять новую память каждый раз, когда мы создаём новый узел. Затем, когда узел освобождается, игре приходится снова освобождать эту память.

Чтобы оптимизировать этот процесс, мы можем использовать пул объектов, который мы обсудим в нижеследующем разделе.

Решение

Эти проблемы загрузки можно решить с помощью шаблона пул объектов (Object Pooling). Пул объектов означает, что мы где-то храним список, также называемый пулом, уже инициализированных узлов. Пример с пуком стре — когда нам нужна стрела, мы можем просто взять одну из этого списка. Когда она больше не нужна, мы возвращаем её в этот список, чтобы её можно было повторно использовать позже.

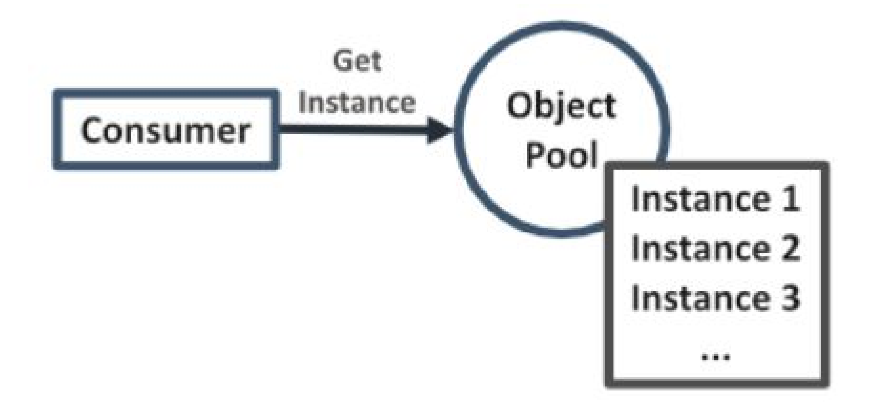


Рисунок 14.2 – Любой класс, которому нужен экземпляр, может

запросить пул объектов

Поскольку мы на самом деле не удаляем и не убираем узел стрелы из дерева сцены, нам нужно будет убедиться через код, что узел перестаёт работать в фоновом режиме, когда он должен быть сохранён в пуле объектов. Когда объект используется, мы говорим, что он живой, потому что он живёт в игре. Когда он находится в пуле объектов, он мёртв, потому что он больше не используется. Когда мы хотим вернуть живой объект в пул, мы говорим, что он уничтожается.

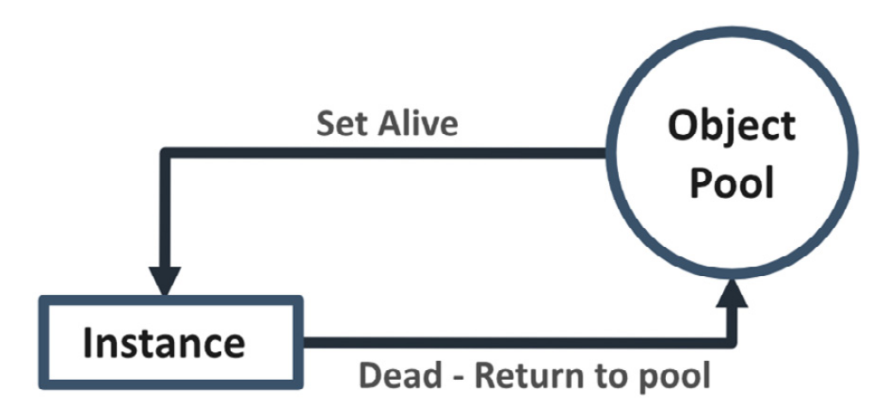


Рисунок 14.3 – Пул объектов устанавливает экземпляр как живой. Когда экземпляр мёртв, он возвращается в пул

Вот пример того, как может выглядеть скрипт пула объектов:

```
class_name ObjectPool extends Object
var _pool: Array
func _init(scene: PackedScene, pool_size: int, root_node
   for _i in pool_size:
      var new_node: Node = scene.instantiate()
      _pool.append(new_node)
      new_node.died.connect(kill_node.bind(new_node))
      root_node.add_child(new_node)
func kill_node(node: Node):
   node.set_dead()
   _pool.append(node)
func get_dead_node() -> Node:
```

```
   if _pool.is_empty():
      return null
   var node: Node = _pool.pop_back()
   node.set_alive()
   return node
func free_nodes():
     for node in _pool:
            node.queue_free()
```

Заметьте, мы сохраняем массив, называемый **_pool**, который будет содержать все наши мёртвые узлы. Сначала мы создаём несколько объектов в функции **_init()** типа **scene** и добавляем эти новые объекты в качестве дочерних к **root_node**, которую мы можем передать этой функции **_init()**. Количество объектов, которыми мы заполняем пул, определяется **pool_size**.

В функции **_init()** мы также подключаемся к сигналу **died** каждого узла с помощью функции **kill_node()**. Это означает, что узел, когда он умирает, должен выдать сигнал **died**. Функции **kill_node()**, в свою очередь, вызовет функцию **set_dead()** на этом узле. Эта функция должна отключить узел и будет отличаться для каждого типа узла, поэтому нам нужно реализовать это позже в определении самого скрипта узла. После этого узел возвращается в **_pool**.

Вы также можете видеть, что я вызвал функцию **bind()** для вызываемого **kill_node – kill_node.bind(new_node)**. Это привязывает аргументы к **Callable**, что означает, что если сигнал испускается и этот **Callable** вызывается, аргументы, которые мы здесь связываем, передаются функции **kill_node()**. Таким образом, мы узнаем, какой объект уничтожается в функции **set_dead()**.

Когда нам нужен экземпляр из пула, мы вызываем функцию **get_dead_node()**, которая сначала проверяет, есть ли еще объекты в пуле; если нет, мы ничего не возвращаем. Если в пуле все еще есть объекты, мы удаляем первый элемент из **_pool**, устанавливаем его как живой, а затем возвращаем его.

Наконец, мы реализовали функцию **free_nodes()**, которая освобождает все узлы, присутствующие в пуле. Таким образом,

мы можем освободить их всех удобным образом, когда останавливаем игру.

Реализация пула объектов в нашей игре

Давайте реализуем пул объектов в нашей собственной игре! Очевидные узлы для объединения в пул из нашей игры, похожей на Vampire Survivor, — это снаряд и враг. Мы будем использовать пул для борьбы со снарядами. Вы всегда можете попробовать создать пул объектов, для врагов:

Создайте скрипт **object_pool.gd**, который имеет точное содержание скрипта из предыдущего раздела. Сохраните его в новой папке — **parts/object_pool**.

Давайте подготовим скрипт **projectile.gd**, чтобы он мог находиться в пуле:

1. Вверху добавьте новый пользовательский сигнал, **died**. Он будет вызван, когда снаряд может вернуться в пул.

   ```
   signal died
   ```

2. Затем добавляем две функции, **set_alive()** и **set_dead()**, которые мы вызываем из пула объектов:

   ```
   func set_alive():
      if multiplayer.is_server():
         set_physics_process(true)
         _enemy_detection_area.monitoring = true
      show()
   func set_dead():
      set_physics_process(false)
      _enemy_detection_area.set_deferred("monitoring",
      hide()
   ```

   Функция **set_alive** включает **_physics_process** и обнаружение столкновений для снаряда, но только если этот код запущен с сервера. Затем он показывает снаряд,

независимо от того, запущены ли мы с сервера или нет, чтобы все могли его видеть. Функция **set_dead** отменяет все эти изменения, чтобы снаряд был непригоден для использования, пока мёртв.

Важное примечание

Мы используем функцию **set_deferred()** в **_enemy_detection_area**, чтобы установить для переменной **monitoring** значение **true** или **false** , поскольку это изменение должно быть включено физическим движком, и нам нужно дождаться выполнения всех физических расчетов для этого кадра. Функция **set_deferred()** устанавливает желаемое нами значение в конце текущего кадра.

1. Теперь замените исходную функцию **_ready()** на функцию из следующего фрагмента кода, которая гарантирует, что новые экземпляры не начнут действовать при их создании и помещении в дерево сцены:

```
func _ready():
   set_dead()
```

2. Наконец, замените упоминания **queue_free()** на **died.emit()**, поскольку теперь пул объектов будет управлять процессом создания узла:

```
func _physics_process(delta: float):
   if not is_instance_valid(target):
      died.emit()
      return
   # Rest of _physics_process
func _on_enemy_detection_area_body_entered(body: No
   body.get_hit()
   died.emit()
```

3. Далее давайте изменим скрипт **main.gd** , чтобы создать пул объектов снарядов.

Вверху добавьте переменную **projectile_pool** и предварительно загрузите сцену **projectile.tscn**:

```
var projectile_pool: ObjectPool
var projectile_scene: PackedScene = preload("res://
```

4. Теперь мы хотим инициализировать эту переменную только при запуске с сервера. Сервер будет управлять всеми снарядами. Добавьте следующую строку в **_ready()**:

```
Func _ready():
   # ...
   if multiplayer.is_server():
      # Код для настройки сервера
      projectile_pool = ObjectPool.new(projectile_s
```

5. Пул объектов не освободится сам, когда мы закроем игру, поэтому нам придется сделать это вручную в функции **_exit_tree()** основного скрипта:

```
func _exit_tree():
   if projectile_pool:
      projectile_pool.free_nodes()
      projectile_pool.free()
```

Важное примечание

Узлы в дереве сцены будут автоматически освобождены, когда мы закроем игру. Но объекты, которые не находятся внутри дерева, такие как **projectile_pool** или узлы, которые мы вынимаем из дерева сцены, не управляются тем же процессом. Поэтому нам нужно самим управлять тем, когда их удалять.

1. Наконец, нам нужно обновить скрипт **player.gd**, чтобы получить доступ к пулу объектов для снаряда и задать его цель и позицию. Замените оригинальный способ создания нового снаряда на этот код:

```
func _on_shoot_timer_timeout():
   # Код прицельной стрельбы по выбранному врагу
   var new_projectile: Projectile = get_parent().pr
   if new_ projectile:
      new_projectile.target = closest_enemy
      new_projectile.position = global_position
```

Это всё, что нам нужно сделать для внедрения нашего пула объектов в нашу многопользовательскую игру. Когда вы посмотрите на **Удалённое дерево (Remote Tree)** во время игры, вы увидите, что в начале было создано 50 снарядов, готовых к запуску игроками.

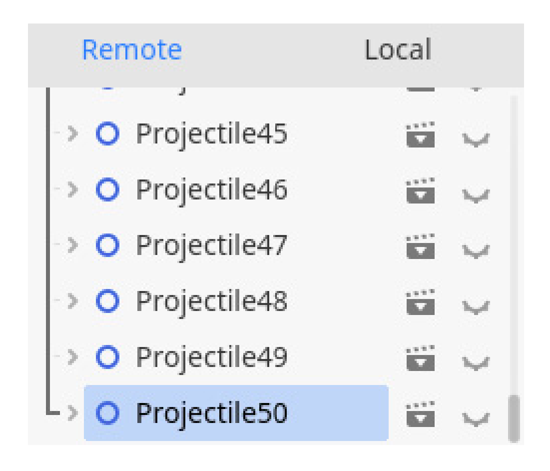


Рисунок 14.4 – Создано 50 снарядов, готовых к использованию.

Вот и всё о шаблоне пул объектов (Object Pool). Он очень полезен для контроля частоты кадров, когда вам нужно, чтобы много объектов появлялось и исчезало часто. Давайте рассмотрим ещё один шаблон в следующем разделе.

Работа с конечными автоматами

Игры — это огромные фрагменты кода, которые могут быть

довольно сложными. Чтобы снизить сложность кода, мы можем попытаться разделить разные фрагменты так, чтобы они выполняли только одно действие очень хорошо. Это именно то, что мы собираемся сделать с помощью конечного автомата. Давайте лучше начнём с постановки проблемы.

Проблема

Агентам, таким как игрок или враги, часто приходится действовать в очень разных сценариях. В платформерной игре, такой как **Super Mario Bros**, например, персонаж должен уметь ходить, бегать, прыгать, нырять, скользить по стенам, летать и так далее. Это много разных видов кода. Если мы попытаемся втиснуть это в один большой класс для игрока, то получим беспорядочный код, который очень сложно понять, отладить или расширить.

В конечном счёте, мы хотим, чтобы код нашей игры был легко понятным и поддерживаемым. Вот почему мы рассмотрим конечный автомат (State Machine) в следующем разделе.

Решение

Отличный способ борьбы с этой сложностью — разделить поведение для каждого из этих желаемых поведений (ходьба, прыжки и т.д.) на разные файлы и классы. Это именно то, что делает шаблон конечный автомат (State Machine). Конечный автомат заменяет часть или всё поведение объекта другим поведением в зависимости от того, в каком состоянии он находится.

Каждое из поведений, которые мы определили ранее (ходьба, прыжки и т.д.), определяется как совершенно независимое состояние, которое изменяет поведение агента и сохраняется в отдельном файле.

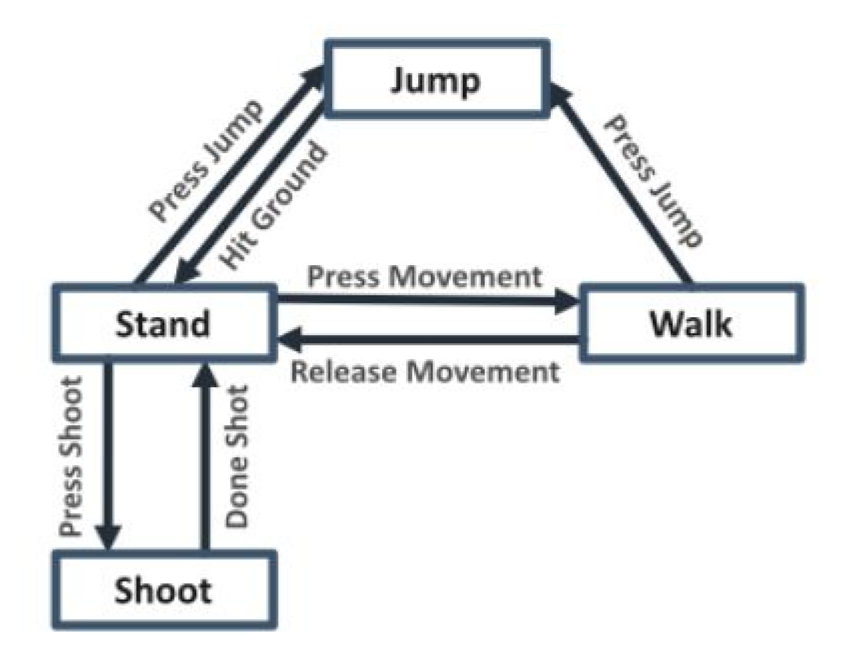


Рисунок 14.5 – Пример того, как состояния могут взаимодействовать друг с другом

Конечный автомат может иметь только одно активное состояние в один момент времени. Это правило гарантирует, что мы не перепутаем поведения или код.

Каждое из этих состояний знает, в какие другие состояния оно может перейти. Этот переход запускается из кода этого состояния, запрашивая машину состояний напрямую для перехода.

Теперь, когда у нас есть поверхностное представление о том, что может делать конечный автомат, давайте быстро перечислим всё, что он должен делать. Конечный автомат должен делать следующее:

- Иметь список всех возможных состояний
- Назначить одно активное состояние
- Уметь переходить из одного состояния в другое
- Обновлять текущее активное состояние и предоставить ему прямой ввод

Имея это в виду, давайте взглянем на код самого конечного автомата:

```
class_name StateMachine extends Node
@export var starting_state: String
var states: Dictionary
var current_state: State
func _ready():
   for child in get_children():
      states[child.name] = child
      child.state_machine = self
   if not starting_state.is_empty:
      transition_to(starting_state)
func transition(state_name: String):
   if current_state:
      current_state.exit()
   current_state = states[state_name]
   current_state.enter()
func _physics_process(delta: float):
   if not current_state: return
   current_state.process(delta)
func _input(event: InputEvent):
   if not current_state: return
   current_state.input(event)
```

Вы можете видеть, что в функции **_ready()** мы сканируем все дочерние элементы конечного автомата и добавляем их в словарь состояний — **states**. Этот словарь поможет нам быстро находить состояния, когда они нам понадобятся в функции **transition()**. Это также означает, что мы добавим каждое состояние как дочерний узел к самомуконечному автомату, например:



Рисунок 14.6 – Конечный автомат, в котором каждое состояние является дочерним узлом

В конце функции **_ready()** мы переходим к **starting_state —** переменной экспорта, которую мы можем использовать для установки начального состояния конечного автомата.

В функции **transition()**, которая используется для перехода в новое состояние, мы сначала проверяем, есть ли у нас **current_state**; если да, то сначала нам придётся вызвать для него функцию **exit()**, чтобы убедиться, что он может очиститься. После этого мы используем **state_name** , который предоставляется в качестве аргумента, для поиска следующего состояния, назначаем его как **current_state**, и вызываем для него функцию **enter()**.

Методы **_physics_process()** и **_input()** используются для прямой передачи данных в функции **process()** и **input() current_state**, если есть текущее состояние.

Теперь давайте рассмотрим сам класс **state**:

```
class_name State extends Node
var _state_machine: StateMachine
func enter():
   pass
func exit():
   pass
func process(delta: float):
   pass
func input(event: InputEvent):
```

```
   pass
```

Класс состояния — это простой скелет с функциями, которые мы должны реализовать при наследовании от него. Это означает, что если у нас есть состояние прыжка, например, нам нужно убедиться, что функции **enter()**, **exit()**, **input()** и **process()** работают так, как и должны, во время выполнения прыжка нашего персонажа.

Если мы хотим перейти из одного состояния в другое, мы можем просто использовать **_state_machine.transition()** из состояния и указать имя состояния, в которое мы хотим перейти.

Теперь мы можем создавать специализированные состояния и связывать их через код, вызывая функцию **transition()** для объекта **_state_machine**.

Пример состояния

Давайте быстро рассмотрим пример состояния, **Walk** для игрока. Это состояние, когда игрок свободно перемещается:

```
extends State
var _player: Player = owner
@export var max_speed: float = 500.0
@export var acceleration: float = 2500.0
@export var deceleration: float = 1500.0
func process(delta: float):
   var input_direction: Vector2 = Input.get_vector("move
   if input_direction != Vector2.ZERO:
   _player.velocity = _player.velocity.move_toward(input
   else:
      _player.velocity = _player.velocity.move_toward(Ve
   _player.move_and_slide()
func input(event: InputEvent):
   if event.is_action_pressed("jump"):
      _state_machine.transition("Jump")
```

Вы можете видеть, что мы расширяем предыдущий скрипт **State** script. Затем мы реализуем функцию **process()** для выполнения наших расчетов движения, которые специфичны для ходьбы, и функцию **input()** для определения того, когда мы хотим перейти из этого состояния в состояние **Jump**.

Нам не нужно переопределять каждую функцию из скрипта **State**, только те, которые нам нужны, в данном случае это функции **process()** и **input()**.

Конечные автоматы, так или иначе, используются почти в каждой игре, в которую вы когда-либо играли. Это очень важная концепция для понимания. Они абстрагируют сложное поведение в отдельные классы, которые легко понять и поддерживать.

Давайте завершим главу несколькими дополнительными упражнениями.

Дополнительные упражнения – Заточка топора

1. Реализация нашей шины событий позволяет подписаться на событие, но не отписаться, когда получатель больше не хочет подписываться. Реализуйте функцию **unsubscribe()**, которая отписывает **Callable** от события:

   ```
   func unsubscribe(event_name: String, callback: Call
       # Ваш код
   ```

2. Конечный автомат, который мы реализовали, ничего не возвращает, когда мы пытаемся вызвать **get_dead_node()**, пока пул пуст. Более разумным способом решения этой проблемы было бы создание нового объекта, по сути, расширение Object Pool на лету. Создайте новую функцию **get_dead_node_or_create_new()** таким образом, чтобы, когда пул пуст, она создавала новый объект, который правильно подключается и возвращается в пул, когда он

умирает.

Итоги

После того, как мы узнали, как программировать и создавать игры, мы наконец сделали шаг назад и узнали о шаблонах более высокого уровня, которые помогают нам структурировать наш проект и код. Сначала мы узнали, что такое шаблоны программирования в целом. Затем мы узнали о шаблонах Шина событий (Event Bus), Пул объектов (Object Pool) и Конечный автомат (State Machine), которые могут помочь нам разными способами. Эти три шаблона являются одними из наиболее широко используемых в играх и применяются также за пределами разработки игр.

Далее вы можете начать изучать более узкоспециализированные шаблоны программирования, например следующие:

- Шаблон **Компонент (Components), так же известный под названием Композиция Composition**: https://gameprogrammingpatterns.com/component.html
- **Команда (Commands)**: https://gameprogrammingpatterns.com/command.html
- **Локатор служб (Service Locators)**: https://gameprogrammingpatterns.com/service-locator.html

В следующей главе мы рассмотрим файловую систему и узнаем, как сохранить состояние нашей игры, чтобы игроки могли начать игру с того места, где они остановились.

Опрос

- Шаблоны программирования — это стандартизированные способы решения проблем в программе или игре. В чём преимущество их знания?
- Любой фрагмент кода можно считать шаблоном. Но когда мы называем что-то антишаблоном или антипаттерном,

означает ли это, что оно работает в нашу пользу?

- Шаблоны «Сигналы» и «Шина событий» очень похожи, поскольку в обоих случаях мы подписываемся на события, но в чём их принципиальное различие?
- Зачем нам использовать шаблон «Конечный автомат» в нашей игре?
- С помощью какой строки кода мы можем перейти из одного состояния в другое, используя шаблон конечного автомата?

15

Использование файловой системы

В ранние дни аркадные игры никогда не сохраняли прогресс игроков. Каждый раз, когда вы вставляли четвертак, игра начиналась с нуля, если только не было системы, которая позволяла бы вам покупать больше жизней в течение того же забега. Но в целом вы не могли вернуться на следующий день и начать играть с того места, где остановились накануне.

Даже ранние консольные игры имели ограниченную функциональность в плане сохранения вашего прогресса. В некоторых играх была система кодов, с помощью которой вы получали секретный код с момента прохождения уровня. Позже вы могли использовать этот код, чтобы начать прямо оттуда. Но эти игры все еще не сохраняли ваш прогресс.

Это ограничение было отчасти связано с тем, что дисковое пространство, такое как жесткие диски или флэш-память, было очень дорогим. В настоящее время почти каждый компьютер и консоль поставляются с несколькими сотнями гигабайт, если не терабайт, памяти. Сохранение данных стало очень дешевым и простым, и игроки привыкли ожидать, что какой-то прогресс отслеживается между игровыми сессиями.

В этой главе мы рассмотрим следующие основные темы:

- Что такое файловая система?
- Создание системы сохранения

Технические требования

Как и в случае с каждой предыдущей главой, окончательный

код можно найти в репозитории GitHub в подпапке для этой главы: https://github.com/PacktPublishing/Learning-GDScript-by-Developing-a-Game-with-Godot-4/tree/main/chapter15

Что такое файловая система?

Файловая система — это система, которая управляет файлами, их содержимым, а также метаданными этих файлов. Например, файловая система будет управлять тем, в каких папках хранятся файлы. Она гарантирует, что мы можем получить доступ к этим файлам для чтения содержимого и метаданных и записи новых данных обратно. Для Godot это означает, что Godot Engine управляет всеми ресурсами, которые могут нам понадобиться в нашей игре, от сцен до скриптов, а также изображений и звуков.

Метаданные

Когда у нас есть данные, например текстовый файл, они часто сопровождаются метаданные (metadata). Это данные о данных. В то время как текстовый файл содержит фактические данные, то есть текст, метаданные содержат информацию, такую как дата создания, кто был автором, где он хранится и какие учётные записи имеют доступ к файлу.

Давайте начнём в следующем разделе изучение файловых систем с путей к файлам.

Пути к файлам

Чтобы иметь возможность найти файл, файловая система даёт уникальный путь к каждому файлу. На нашем компьютере мы можем находить файлы через папки, также называемые каталогами, где мы храним их в удобном порядке. Этот путь может выглядеть так в системе на базе Windows:

```
C:\Users\user_name\Documents\my_text_file.txt
```

Или это может выглядеть так в системах на базе macOS и Linux:

```
~/Documents/my_text_file.txt
```

Для файлов ресурсов и других файлов, связанных с проектом, пути Godot Engine работают относительно позиции, где находится файл **project.godot** Путь этого файла считается корневым каталогом. Пути в файловой системе Godot для доступа к файлам ресурсов всегда начинаются с **res://**. Например, для доступа к одному из файлов в проекте путь может выглядеть следующим образом:

```
res://parts/player/player.tscn
```

Важное примечание

Для удобства и совместимости файловая система Godot всегда использует прямые слеши (/). Даже в системах на базе Windows, где обычно используется обратный слеш (\).

На самом деле мы уже использовали один из этих путей, когда предварительно загружали снаряды в *Главе 10* .

Путь пользователя

Тот факт, что мы можем легко получить доступ ко всем файлам проекта, используя путь **res://**, очень удобен, но есть проблема. Мы не можем записывать ни в один файл в домене **res://**; когда игра запущена из экспортированной сборки, мы можем только читать из неё файлы. Чтобы помочь разработчикам с этой проблемой, Godot Engine предоставляет еще один корневой путь, **user://**, в который можно записывать файлы, а затем читать из него.

Godot Engine автоматически создаёт папку где-то на компьютере для хранения этих пользовательских данных. Расположение этой папки зависит от системы, на которой запущена игра, поэтому для каждой ОС она будет разной:

- Windows: **% APPDATA%\Godot\app_userdata\<имя_проекта>**
- macOS: **~/Library/Application Support/Godot/app_userdata/ < имя_проекта>**
- Linux: **~/.local/share/godot/app_userdata/ < имя_проекта>**

Важное примечание

В настройках проекта мы даже можем указать, где разместить эту папку для каждой из трех ОС, но сейчас это не нужно, так как Godot сделает это за нас и спрячет папки в надежном месте.

Вы можете получить доступ к папке **user://** для определенного проекта, открыв меню **Проект (Project)** и выбрав **Открыть папку пользовательских данных (Open User Data Folder)**.

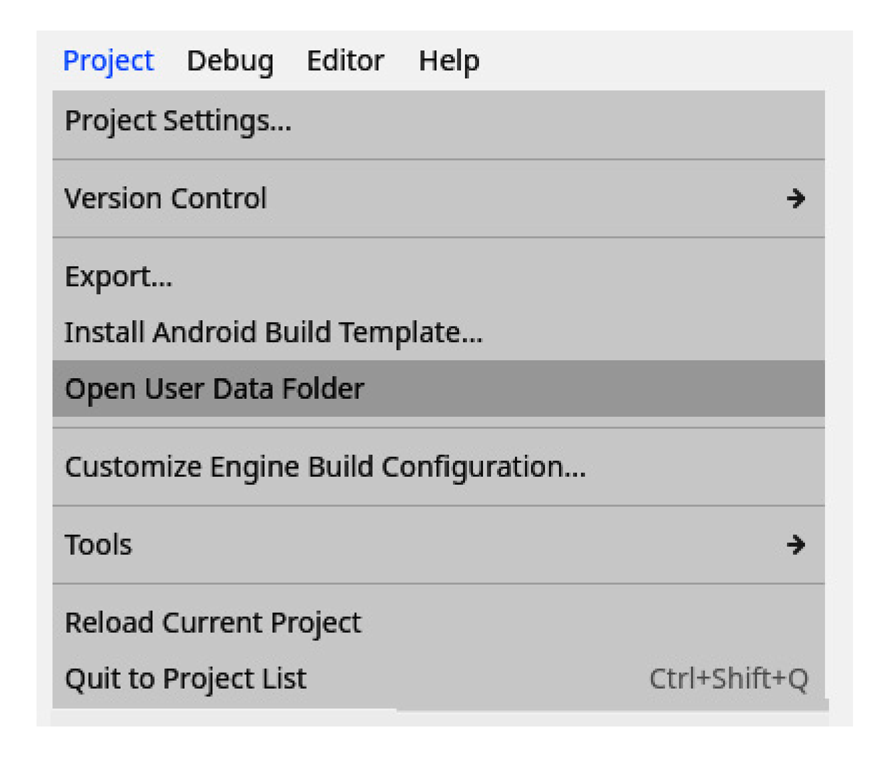


Рисунок 15.1. Открытие папки пользовательских данных переносит нас в папку user://

Путь **user://** — это то, что мы собираемся использовать для записи наших данных сохранения в следующем разделе этой главы. Итак, давайте приступим к фактической реализации нашей собственной небольшой системы сохранения.

Создание системы сохранения

Теоретически, всё, что нам нужно сделать, это открыть файл, записать в него данные, которые мы хотим сохранить, а затем, позже, читать этот же файл, когда нам нужны данные. Как оказалось, в Godot Engine действительно легко читать и записывать файлы.

Сначала мы рассмотрим, как записывать данные во внешний файл.

Запись данных на диск

Давайте приступим к этому, создав новый скрипт под названием **save_manager.gd** в папке **autoloads**. Затем, чтобы сохранить данные, поместите этот код в скрипт:

```
extends Node
const SAVE_FILE_PATH: String = "user://save_data.json"
var save_data: Dictionary = {
   "highscore": 0
}
func write_save_data():
   var json_string: String =
    JSON.stringify(save_data)
   var save_file: FileAccess =
    FileAccess.open(SAVE_FILE_PATH, FileAccess.WRITE)
   if save_file == null:
      print("Could not open save file.")
      return
```

```
save_file.
 store_string(json_string)
```

В верхней части скрипта мы определяем словарь, называемый **save_data**, который мы будем использовать для хранения всех данных. На данный момент он содержит только **highscore**. Если мы захотим получить доступ к сохраненным данным позже во время игры, мы можем просто использовать эту переменную.

Кроме того, у нас есть функция **write_save_data()**есть функция **save_data** в строку JSON с помощью функции **JSON.stringify()**.

Стандарт JSON

JSON — это формат данных, который широко используется в Интернете и на других платформах. Название **JSON** означает **нотация объектов JavaScript (JavaScript Object Notation)**.Это очень легкий способ хранения данных, который имеет дополнительное преимущество в том, что его легко читать и корректировать после сохранения данных в файле.

Далее мы используем класс **FileAccess** для открытия файла, в который мы хотели бы записать наши данные. Мы сохранили путь к файлу как константу **SAVE_FILE_PATH** в верхней части скрипта. Поскольку мы хотим записать в файл, нам нужно открыть его с правами записи, предоставив **File.Access.WRITE** для функции **open()**. Этот режим доступа к файлу также создаст файл для нас, если он еще не существует. Открытый файл сохраняется в переменной **save_file**.

Затем мы проверяем, правильно ли открылся **save_file**. Если по какой-либо причине файл не удалось открыть, значение этой переменной будет равно **null** и нам следует прекратить выполнение функции.

Дополнительная информация

Более подробную информацию о классе **FileAccess** можно найти в официальной документации: https://docs.godotengine.org/en/stable/classes/class_fileaccess.html.

Последнее, что нам нужно сделать, когда файл правильно открыт, это записать в него данные JSON. Мы делаем это с помощью **store_string()** в **save_file**, передавая **json_string**.

Это все, что нам нужно сделать для записи данных в файл в папке **user://**. Теперь мы можем написать функцию, которая считывает эти данные обратно.

Чтение данных с диска

Для чтения данных из папки **user://** мы выполняем те же шаги, что и для записи, но в обратном порядке. Добавьте эту функцию в скрипт **save_manager.gd**:

```
func read_save_data():
   var save_file: FileAccess = FileAccess.
    open(SAVE_FILE_PATH, FileAccess.READ)
   if save_file == null:
      print("Не удалось открыть файл сохранения.")
      return
   var file_content: String = save_file.
    get_as_text()
   save_data =
    JSON.parse_string(file_content)
```

Функция **read_save_data** загружает сохраненный файл и анализирует его содержимое, чтобы мы могли использовать его в игре. Сначала мы открываем сохранённый файл с помощью **FileAccess.open**, предоставляя путь к файлу и **FileAccess.READ**, чтобы указать, что мы хотим только прочитать его. После этого мы проверяем, что файл открыт правильно, в противном случае нам нужно снова выйти из функции.

Далее мы считываем весь файл как строку в переменную **file_content**. Нам придется разобрать эту строку из формата JSON, в котором она была сохранена, в формат, который может обрабатывать GDScript, словарь. Разобранное значение напрямую сохраняется в переменной **save_data**, которую мы

определили в предыдущем разделе.

Дополнительная информация

Дополнительную информацию о сохранении и загрузке данных в Godot Engine можно найти в официальной документации: https://docs.godotengine.org/en/stable/tutorials/io/saving_games.html.

Отлично, у нас есть две функции, которые могут записывать и читать сохранённые данные для нашей маленькой игры. Нам всё ещё нужно добавить некоторые функции, чтобы убедиться, что скрипт может использоваться игрой.

Подготовка менеджера сохранений для использования в игре

Менеджер сохранений почти готов, но нам еще нужно добавить эти две функции:

```
func _ready():
   read_save_data()
func save_highscore(new_highscore: float):
   save_data.highscore = new_highscore
   write_save_data()
```

Первая функция, функция **_ready()**, обеспечивает загрузку сохранённых данных с момента запуска игры игроком.

Вторая функция добавляет удобный способ сохранения нового рекорда. Она добавляет новый рекорд в словарь **save_data**, а затем записывает данные на диск.

Теперь, чтобы убедиться, что мы можем получить доступ к менеджеру сохранений из любого места, добавьте этот скрипт в автозагрузки проекта. Мы хотим, чтобы наш менеджер сохранений был первой автозагрузкой, которая будет выполнена, что гарантирует загрузку сохранённых данных до выполнения любой другой части игры. Для этого убедитесь, что

скрипт **save_manager.gd** находится в верхней части списка автозагрузок. Вы можете сделать это, перетащив запись **SaveManager** или нажимая на стрелки справа, пока она не окажется наверху.

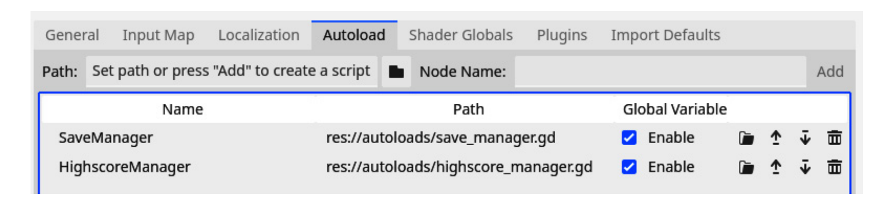


Рисунок 15.2. Убедитесь, что SaveManager является первой автозагрузкой в списке

Закончив этот скрипт и разместив его в качестве автозагрузки, мы наконец можем подключить игру для его использования. Давайте сделаем это в следующем разделе.

Настройка игры для использования менеджера сохранений

Теперь все, что нам нужно сделать, это получить highscore из **SaveManager**, когда мы загружаем скрипт **highscore_manager.gd** и сохранять достижение каждый раз, когда игрок достигает нового. Добавьте следующую функцию **_ready** в скрипт **highscore_manager .gd** и добавьте вызов функции **SaveManager.save_highscore()**:

```
func _ready():

    highscore = SaveManager.save_data.highscore
func set_new_highscore(value: int):
   if value > highscore:
      highscore = value

    SaveManager.save_highscore(highscore)
```

После этого мы наконец-то можем немного поиграть в игру,

набрать наибольшее количество очков, закрыть игру и, когда мы снова её откроем, увидеть наш предыдущий рекорд.



Рисунок 15.3 — Рекорд загружается, когда мы открываем игру

Теперь наша игра действительно готова к тому, чтобы пользователи стремились побить свой рекорд за несколько дней без необходимости следить за ним самостоятельно. В следующем разделе мы взглянем на фактическое содержание самого файла сохранения.

Просмотр файла сохранения

На данный момент мы относимся к файлу сохранения как к черному ящику, не зная его фактического содержимого. Мы сохраняем в нем данные в формате JSON и считываем их обратно, разбирая их обратно на данные, которые можно использовать в GDScript.

Чёрный ящик

Мы говорим, что взаимодействуем с чёрным ящиком, когда не имеем представления о том, как на самом деле работает эта штука. Мы даём системе ввод, а она выдает какой-то вывод.

Конечно, мы также можем просмотреть содержимое файла сохранения с помощью текстового редактора, например, Notepad в Windows. Просто откройте папку **user://**, как мы делали в разделе *Путь пользователя* ранее в этой главе. Отсюда откройте файл **save_data.save**, который мы создали в менеджере сохранения.

Вы увидите, что данные в этом файле очень легко читаются и выглядят очень похоже на реальный словарь, который мы определили в скрипте **save_manager.gd**. Это потому, что JSON также имеет понятие структуры данных словаря, а синтаксис очень похож на синтаксис словарей в GDScript. Файл выглядит так:

```
{"highscore":56}
```

Если хотите, можете изменить данные сохранения отсюда и схитрить, вписав невозможно высокий рекорд. К сожалению, пользователи тоже смогут сделать это, если будут знать, где искать файл сохранения.

Дополнительная информация

Существуют способы шифрования файлов сохранения, но они выходят за рамки этой книги. Подробнее см. в официальной документации: https://docs.godotengine.org/en/stable/classes/class_fileaccess.html#class-fileaccess-method-open-encrypted.

Теперь, когда наша игра сохраняет рекорд игрока, мы подошли к концу этой главы. Есть ещё много трюков, которые нужно изучить, чтобы загрузить и сохранить состояние игры, но на данный момент этого, безусловно, будет достаточно.

Итоги

В этой главе мы узнали всё о файловой системе Godot и компьютера в целом. Это позволило нам написать небольшую систему сохранения, которая хранит рекорд нашей игры и загружает его каждый раз, когда мы запускаем игру.

Это была последняя глава этой части книги. В течение последних пяти глав мы глубже погрузились в концепции программирования, шаблоны и файловую систему.

Вы все готовы пойти и разработать несколько собственных игр. Но прежде чем вы это сделаете, я хочу дать вам несколько последних указаний и шагов о том, что делать дальше в последней главе этой книги. Увидимся там.

Опрос

- В чём разница между путями к файлам **res:**// и **user:**// в Godot Engine?
- Для сохранения данных мы использовали формат JSON. Является ли JSON уникальным форматом Godot Engine? В какой еще области широко используется формат JSON?

16

Что дальше?

Какое путешествие! Мы прошли путь от полного отсутствия знаний о программировании до промежуточного понимания и создания целой игры с нуля в рамках одной книги. Я знаю, что не всегда легко понять всё с первого раза, но именно так мы учимся. Мы должны терпеть неудачу в чем-то и пробовать это снова и снова – каждый раз становясь лучше в том, чему пытаемся научиться.

Начало нового проекта — это всегда борьба. Это как белый холст художника или чистая страница писателя. Начинать что-то новое трудно. Вот почему я хотел бы дать вам несколько идей, чтобы вы начали. И не беспокойтесь о том, какой проект является идеальным. С этого момента любой проект, который вас интересует, хорош для пробы. Просто убедитесь, что они небольшие и закончите их, чтобы люди могли играть в то, что вы сделали.

В этой главе мы рассмотрим следующие основные темы:

- Идеи для вашего следующего проекта
- Изучение новых тем
- Присоединение к сообществу

Идеи для ваших следующих проектов

Как упоминалось во введении к этой главе, страх перед пустым проектом — это реальная вещь. С чего начать и что делать — это всегда борьба. Давайте подробнее рассмотрим обе проблемы.

Начало нового проекта

Вот несколько советов, которые я усвоил на собственном горьком опыте, когда дело касается начала нового проекта. Они помогут вам сосредоточиться на создании игры, которую вы хотите создать:

- **Начните с создания основного игрового цикла**: Первое, что вам следует сделать, это создать игровой процесс, на котором будет основана игра. Если вы создаёте платформер, убедитесь, что перемещение и прыжки сами по себе интересны, прежде чем вы начнёте добавлять более сложные системы.
- **Сделайте его функциональным, прежде чем сделать его красивым**: легко запутаться, делая игру слишком красивой, но это значительно замедлит производство игры. Хуже того, если игра неинтересная и вам нужно переделать много систем, вам также придется переделывать всё то, что вы сделали, чтобы сделать её красивой.
- **Сосредоточьтесь на том, что вы хотите изучить**: если есть конкретная тема, которую вы хотите изучить, например, меню и пользовательский интерфейс, то сделайте это фокусом своего проекта. Есть много игровых жанров, ориентированных на пользовательский интерфейс, например, визуальные новеллы или стратегические игры, которые вы можете создать.
- **Оставьте это небольшим (Keep it small)**: большинство людей предпочитают играть в небольшую, веселую игру, нежели в длинный, полусырой проект. Это также гарантирует, что вы сможете закончить игру. Любая игра, в которую вы когда-либо играли, была завершена (или считалась достаточно завершённой), в то время как в незаконченные игры никто никогда не играл.
- **Создайте дизайн-документ игры (GDD)**: в начале проекта создайте краткий документ по дизайну игры. Это может быть одностраничный документ, просто чтобы изложить общие намерения проекта. Он будет направлять вас во время разработки и поможет принимать решения в дальнейшем.

Это были некоторые общие советы о том, как взяться за любой новый проект. Легко увязнуть в деталях, поэтому самое главное — просто начать. В конце концов, обучение происходит, когда мы совершаем ошибки. Теперь перейдём к более конкретным идеям о том, что делать дальше.

Расширение игры на выживание

Логичным следующим шагом будет расширение игры, которую мы создаём на протяжении этой книги. Некоторые идеи в развитие проекта:

- **Представьте новые типы врагов**: это может быть враг, который медлителен, но наносит большой урон, или бомба, которая остаётся на месте и взрывается через 5 секунд.
- **Создйте различные типы снарядов**: например, кинжалы с меньшей дальностью полета, но наносящие больше урона врагам.
- **Добавьте больше коллекционных предметов**: к примеру, щит, который защищает игрока в течение пяти секунд, или ботинки, которые позволяют игроку быстрее передвигаться.

Однако одно предупреждение, которое я вам дам, заключается в том, что вам не следует слишком долго зацикливаться на одном проекте. Когда вы учитесь создавать и проектировать игры, лучше всего делать небольшие проекты и не дорабатывать одну игру слишком сильно. Лучше считать что-то готовым, дать людям поиграть в это, а затем перейти к следующему проекту — это до тех пор, пока вы не почувствуете себя достаточно уверенным в своих навыках, чтобы взяться за более крупную игру.

Полировка видеоигр

В разработке видеоигр мы называем **полировкой** всё, что не является основным опытом игры– вещи, которые не являются неотъемлемой частью опыта, но делают его более красивым или более плавным. Полировка очень важна для

окончательного опыта игры.

Продолжая тему создания множества небольших игр, давайте рассмотрим несколько идей для игр, над которыми вы могли бы поработать.

Создание ещё одной игры

В сообществе разработчиков игр мы часто говорим, что *ваши первые 10 игр будут отстойными*, что является грубым способом сказать, что первые 10 проектов, которые вы создадите, не будут отличными. И это нормально. Мы все через это прошли. Но каждый проект, который вы получите, научит вас чему-то.

Возможно, в одном проекте вы сосредоточитесь на создании увлекательного игрового цикла, а в другом — на создании понятных меню.

Постарайтесь реализовать много небольших проектов, каждый из которых научит вас чему-то полезному; таким образом, вы сможете развивать свои навыки, а также иметь что-то, что можно будет продемонстрировать.

Вот несколько идей для игр, которые вы можете попробовать реализовать:

- **Платформер** : научитесь работать с 2D-физикой, где гравитация тянет игрока вниз.
- **Пошаговая стратегическая игра**: узнайте, как разделить игровой процесс на отдельные этапы.
- **Карточная игра**: узнайте, , как работать с более сложным пользовательским интерфейсом и как абстрагировать способности карт.
- **Игра-головоломка**: научитесь придумывать интересные головоломки, которые не будут слишком сложными и слишком простыми.
- **Визуальная новелла**: узнайте всё о меню и о том, как интегрировать диалоги в игру.
- **Бесконечный раннер**: узнайте, как генерировать

случайные уровни на лету.

Работая над одним из таких проектов, найдите части, которые вы не знаете, как сделать или не понимаете. Информация в этой книге — отличная отправная точка, но её будет недостаточно, чтобы решить всё. Книга, которая попытается сделать это, будет громоздкой и длинной.

Бесплатные игровые ресурсы

Если вы не художник или музыкант, то создание ресурсов для вашей игры может быть сложным. К счастью, есть много бесплатных вариантов; проверьте эти ссылки:

- **Kenney**: потрясающий бесплатный 2D, 3D и аудио актив, созданный в Нидерландах. Спрайты, используемые в этой книге, взяты из Kenney: https://kenney.nl/assets.
- **OpenGameArt**: сообщество разработчиков и художников игр, предоставляющее бесплатные ресурсы с открытым исходным кодом: https://opengameart.org/.
- **Itch.io**: платформа, на которой мы опубликовали нашу игру, также является отличным ресурсом для бесплатных ресурсов всех видов: https://itch.io/game-assets/free.

С их помощью вы сможете создать прекрасную игру в кратчайшие сроки.

Несмотря на то, что эти ресурсы бесплатны, часто требуется указать создателя ресурсов где-то в игре или на странице магазина игры. Так что не забывайте это делать; таким образом, вы получаете бесплатные ресурсы, а они получают известность. Все выигрывают!

Знание того, что изучать, — это первый шаг, который вы, как мы надеемся, определите, выполняя проекты, предложенные в этом разделе. Отсюда вопрос: как мы изучаем эти вещи? Давайте рассмотрим это в следующем разделе.

Изучение новых тем

Существует множество отличных ресурсов, чтобы узнать больше о программировании и игровом дизайне. В этом разделе мы рассмотрим несколько различных типов ресурсов, которые могут вам помочь.

Следование конкретным руководствам

Когда вы знаете, что именно вы хотели бы изучить, наверняка найдется обучающее видео или текстовое руководство по этой теме. Все, что вам нужно сделать, это воспользоваться вашей любимой поисковой системой, например Google или YouTube, и найти нужную вам тему. Вот некоторые из моих любимых источников отличных обучающих программ Godot Engine:

- **GDQuest**: Здесь представлены великолепные видеоуроки по широкому кругу тем и даже целые курсы: https://www.youtube.com/@Gdquest.
- **Hearbeast**: канал на YouTube с полными игровыми проектами, а также краткими обучающими материалами по различным темам, связанным с Godot Engine: https://www.youtube.com/@uheartbeast.
- **The Godot Engine docs**: в документации есть целая страница, которая знакомит с различными сторонними руководствами: https://docs.godotengine.org/en/stable/community/tutorials.html.
- **KidsCanCode**: здесь вы найдете множество замечательных статей на различные темы, связанных с Godot: https://kidscancode.org/godot_recipes/4.x/.
- **Game Maker's Toolkit**: Канал с отличными видеоэссе о дизайне игр: https://www.youtube.com/@GMTK.

Недостатка в онлайн-уроках по движку Godot нет.

Читайте больше книг

Другой способ углубить свои знания — читать другие книги по более специализированным и продвинутым темам. Вот некоторые из моих любимых:

- *Искусство игрового дизайна: Книга линз (The Art of Game Design: A Book of Lenses)*, Джесси Шелл (Jesse Schell). Эта книга — очень подробное введение во все, что связано с игровым дизайном.
- *Шаблоны игрового программирования (Game Programming Patterns)*, Роберт Нистром (Robert Nystrom). Хотите узнать больше о шаблонах программирования, применяемых в играх? Тогда эта книга для вас.
- *Теория веселья в игровом дизайне (Theory of Fun for Game Design)*, Раф Костер (Raph Koster). Короткая книга о том, что делает игры веселыми и об их культурном значении.
- *Проекты по разработке игр Godot 4 (Godot 4Game Development Projects)*, Крис Брэдфилд Chris Bradfield).
- *GAMEDEV: 10 шагов к успешной первой игре (GAMEDEV: 10 Steps to Making Your First Game Successful)*, Влад Маргулис (Wlad Marhulets).

Любой из них поможет вам на пути к становлению разработчиком и дизайнером игр.

Чтение документации Godot Engine

Наряду со сторонними ресурсами, конечно же, есть и официальная **документация Godot Engine**. Это очень обширный и исчерпывающий источник информации обо всех различных классах и узлах, содержащий руководства по всем различным подсистемам, связанным с движком.

Доступ к документации можно получить здесь: https://docs.godotengine.org/.

Всякий раз, когда вы ищете, как использовать определённую часть двигателя, вам следует начать с изучения документации.

Просмотр игрового кода чужих проектов

Отличный способ учиться — смотреть на игровые проекты и код других людей. Поскольку Godot Engine — проект с

открытым исходным кодом, многие игры на этом движке также имеют открытый исходный код. Это означает, что полный проект доступен онлайн для всех желающих, чтобы посмотреть и поиграться с ним.

Вот некоторые замечательные проекты:

- **Демонстрационные проекты Godot Engine**: это небольшие проекты с полностью открытым исходным кодом, которые варьируются от технических демонстраций до полноценных игр: https://github.com/godotengine/godot-demo-projects.
- **Открытая ролевая игра GDQuest**: демоверсия 2D-ролевой игры с открытым миром: https://github.com/gdquest-demos/godot-open-rpg.

Обратите внимание, что эти проекты часто всё еще находятся в стадии разработки или могли быть реализованы на более старой версии Godot Engine, но не помешает взглянуть на то, как люди решали определённые проблемы.

Изучение новых тем — отличный способ улучшить разработку игр; это позволит вам делать больше и быстрее. Однако ещё один важный, но упускаемый из виду способ улучшить свои навыки — присоединиться к сообществу единомышленников, которые любят делиться, давать отзывы и поддерживать друг друга. Давайте посмотрим, как мы могли бы присоединиться к такому сообществу.

Присоединение к сообществу

Наконец, я хотел бы призвать вас присоединиться к сообществу разработчиков игр. В целом, это очень приветливая и дружелюбная группа людей, которые любят слышать о вас и ваших играх. Не стесняйтесь обращаться к людям и задавать им вопросы. Большинство людей очень отзывчивы и будут поддерживать вас на вашем пути.

Присоединяйтесь к форуму, Discord,

Reddit или любой другой платформе

Существует множество различных сообществ, связанных с созданием игр на разных платформах. Неважно, в каких социальных сетях вы наиболее активны; там, скорее всего, есть оживленное сообщество разработчиков игр. Вот некоторые из каналов, к которым вы можете присоединиться:

- **Официальный форум Godot Engine**: это идеальное место, чтобы задавать вопросы, помогать другим и демонстрировать свои проекты: https://forum.godotengine.org/.
- **Официальный Discord Godot Engine**: здесь вы можете встретить единомышленников-разработчиков игр, которые работают с Godot: https://discord.com/invite/4JBkykG.
- **Godot Engine Reddit**: Здесь вы можете поделиться своим проектом и посмотреть на прогресс других людей: https://www.reddit.com/r/godot/.
- **Разработчики Godot на Twitch**: Если вы увлекаетесь стримингом или смотрите стримы, то есть тег **Godot**, который вы можете проверить: https://www.twitch.tv/directory/all/tags/godot.
- **Любая из социальных платформ**: Многие люди публикуют посты на таких социальных платформах, как **Instagram**, **Twitter**, **Mastodon**, **Blusesky** или **TikTok**. Просто посмотрите посты с хэштегами **#GodotEngine**, **#IndieDev** и **#GameDev**.

Возможность делиться своим прогрессом, наблюдать за прогрессом других людей и обсуждать разработку игр с другими людьми может стать очень мотивирующим и приносящим удовлетворение способом создания небольшого сообщества вокруг вас.

Вклад в проект Godot Engine

Поскольку Godot Engine имеет открытый исходный код, и теперь вы знаете, как с ним работать, вы можете помочь, если

хотите. Вот несколько способов, которыми вы можете это сделать:

- **Список проблем**: Godot Engine, как и любое программное обеспечение, к сожалению, не лишено ошибок, но есть много людей, которые их решают. Если вы нашли ошибку в движке, вы всегда можете открыть issue на странице GitHub для движка: https://github.com/godotengine/godot/issues.
- **Написание документации**: всякий раз, когда вы читаете документацию и чувствуете, что на странице, на которой вы находитесь, чего-то не хватает, не стесняйтесь добавлять свои собственные улучшения и помогать всем, кто ещё не знаком с Godot Engine. На каждой странице документации есть ссылка, которая приведёт вас прямо туда, где вы можете редактировать страницу.

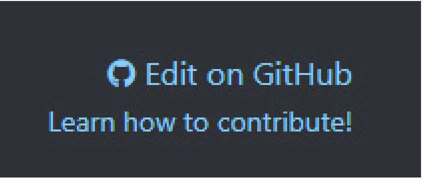


Рисунок 16.1. Нажмите Edit на GitHub, чтобы предложить изменение на странице документации Godot.

- **Перевод частей движка**: хотите помочь локализовать движок или его документацию на другие языки, чтобы больше людей получили доступ к этому замечательному инструменту? Если да, вы найдете всё, что вам нужно знать, здесь: https://docs.godotengine.org/en/latest/contributing/documentation/editor_and_docs_localization.html#doc-editor-and-docs-localization.
- **Вносите код в движок напрямую**: как только вы почувствуете себя достаточно уверенно в своих способностях кодирования, вы можете попробовать кодирование непосредственно в ядре движка. Вы можете найти больше информации здесь: https://docs.godotengine.org/en/latest/contributing/development/index.html.

Это некоторые из самых известных способов, которыми вы можете внести свой вклад в проект и сообщество Godot. Давайте вернёмся к созданию игр в следующем разделе.

Присоединяйтесь к игровому джему

Большая часть работы разработчика игр — это, конечно, разработка игр. И нет ничего более увлекательного, чем разрабатывать игры с вместе другими. Именно для этого и существуют **игровые джемы (game jams)**. Игровые джемы — это мероприятия, на которых вы создаёте игру с нуля за небольшой промежуток времени, часто за выходные, но это может занять месяц или больше.

Цель состоит в том, чтобы создать небольшую игру, в которую затем играют другие участники джема. Это отличный способ получить много отзывов и поиграть в игры многих других людей во время джема. Более того, вы можете участвовать в этих джемах с командой. Таким образом, вы создаете связи в мире разработки игр и учитесь работать с несколькими людьми над чем-то. Однако вы также можете просто участвовать самостоятельно.

Одной из крупнейших платформ, предоставляющих игровые джемы, является itch.io, которую мы использовали для загрузки нашей игры в *Главе 12* . Просто зайдите на https://itch.io/jams и найдите игровой джем, который вас интересует. В любой момент времени всегда проходит много игровых джемов.

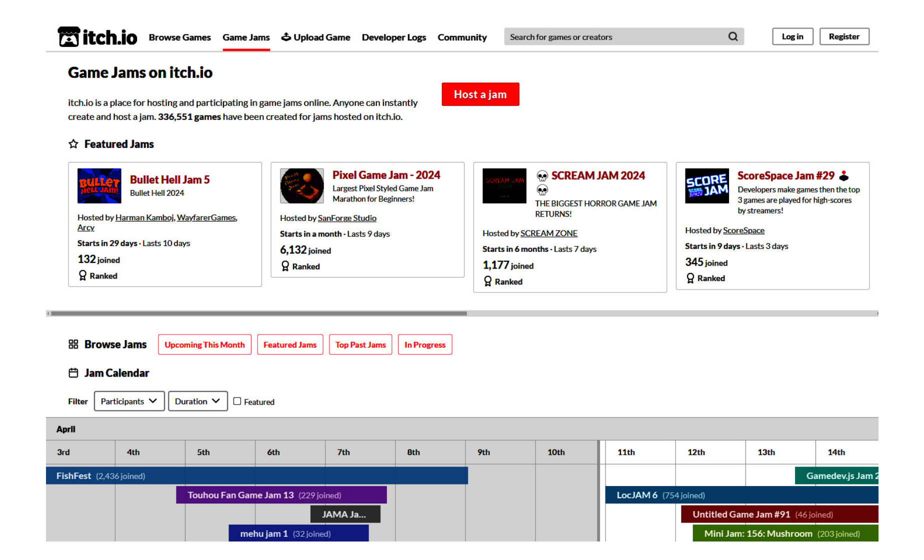


Рисунок 16.2 — Страница игровых джемов на Itch.io

Вот несколько замечательных джемов, к которым вы могли бы присоединиться:

- **Godot Wild Jam**: ежемесячный игровой джем специально для разработчиков игр Godot: https://godotwildjam.com/.
- **Global Game Jam**: физический игровой джем, организованный по всему миру. Посетите сайт, чтобы найти место рядом с вами: https://globalgamejam.org/.
- **The GMTK Game Jam**: ежегодный игровой джем, организованный каналом **Game Developer's Toolkit** на YouTube. У этого нет постоянного веб-сайта, связанного с ним, но он всегда анонсируется на канале YouTube: https://www.youtube.com/@GMTK.

Теперь вы знаете, как интегрироваться в сообщество разработчиков игр и начать делиться своими потрясающими играми с тысячами других единомышленников.

Завершение

В этой главе мы завершили книгу, показав вам, что будет дальше на вашем пути к созданию отличных игр. Мы увидели,

над какими возможными проектами мы могли бы работать дальше, как узнать что-то новое и как присоединиться к сообществу.

Целью всей этой книги было заставить новых людей начать создавать игры на движке Godot Engine. Мы прошли путь от полного отсутствия знаний о программировании или разработке игр до очень прочного и среднего понимания того и другого.

В начале мы узнали, как использовать GDScript в качестве языка по выбору, изучая основные концепции, такие как переменные и циклы, и продвигаясь к более промежуточным темам, таким как классы и надлежащие руководящие принципы программирования. В конце концов мы добрались до продвинутых тем, таких как ключевое слово super и шаблоны программирования.

В ходе работы над книгой мы также разработали собственную игру, небольшую игру на выживание, смоделированную по образцу *Vampire Survivors*. Мы увидели, что такое узлы и какие из них следует использовать в самых разных ситуациях. Мы даже сделали игру многопользовательской и опубликовали её на **Itch.io**, для всех желающих поиграть!

Надеюсь, вы отлично провели время, изучая, как программировать и использовать Godot Engine, и я хотел бы поблагодарить вас от всего сердца за то, что вы остались до конца. Теперь идите и создавайте потрясающие игры!

Алфавитный указатель

As this ebook edition doesn't have fixed pagination, the page numbers below are hyperlinked for reference only, based on the printed edition of this book.

Symbols

2D coordinate system 146, 147

_ready() method 74

@tool annotation 305-307

reference link 307

risks 306

A

abstraction 93, 104

advantages 94

private member 93

public member 93

Android

game, exporting to 295

anti-pattern 311

application layer 248-250

client-server network 251

peer-to-peer network 250, 251

Area2D node 186

arguments 74, 75

arrays 54

creating 54

data types 56

elements, accessing backward 55, 56

elements, modifying 56

manipulating 57, 58

strings, working 57

values, accessing 55

autoloads

highscores, storing in 239

in remote tree 241

using 239

axe

sharepening 178

B

base networking code

client-server connection, creating 253

multiple debug instances, running 255-257

setting up 253

UI, adding 254, 255

Boolean logic 45-48

match statement 48, 49

ternary-if statement 50

break keyword 65, 66

breakpoint 166-168

bugs 60

C

camelCase 102

camera

Camera2D node, setting up 180

drag margins, adding 180, 181

making, that follows player 180

movement, considerations 181-183

movement, smoothing out 183, 184

CharacterBody2D 186

class

defining 79

extending 80, 81

instancing 79, 80

naming 80

script 81

variable 82-84

classes 78

client-server network 251

client synchronization 257

multiplayer scene, updating 257

code

errors and warnings 58-60

collectibles

code, writing for 196, 197

connecting, to signal 194, 195

creating 191

inheriting, from base scene 192-194

layers and masks 198

layers and masks, using 198

layers, naming 198-200

right layers, assigning 200, 201

scene, creating 191

collisions 185

Area2D node 186

physics bodies 185

shape, adding to player node 187-189

static bodies, creating for boulders 189

static bodies, creating for walls 190, 191

collisions, physics bodies

CharacterBody2D 186

RigidBody2D 185, 186

StaticBody2D 186

comments 43

computer networking 248

application layer 250

in Godot Engine 251

IP address 252

port numbers, using 252, 253

transport layer 248, 249

consoles

game, exporting to 295

constants 35

in GDScript 36, 37

magic numbers 37

constructor 81

container 54

container nodes 205

continue keyword 64

control flow

commenting, in code 43

elif statement 42

if-else statement 41, 42

if statement 39-41

indentation 43-45

using 39

D

data

reading, from disk 329

writing, to disk 328, 329

data types 84

floats 34

in arrays 56

integers 34

strings 35

defensive programming 105

dictionaries 66

creating 66

data types 67

functions 68

key-value pair, creating 68

looping through 68

nested loops 69, 70

values, accessing 67

values, modifying 67

direction 148

distance_to() function 153

documentation 109

class documentation, accessing 109, 110

function, defining 111

function documentation, accessing 111

online documentation, accessing 112, 113

searching 112

variable, defining 111

variable documentation, accessing 111

don't repeat yourself (DRY) 105

duplicate() function

used, for duplicating arrays or dictionaries 305

E

elements

accessing, backward 55, 56

elif statement 42

encapsulation 94, 104

end game

fixing 269

synchronizing 270, 271

enemies

base scene, constructing 212

Game Over screen, making 229

making 211

navigating 212, 213

NavigationAgent2D node, adding to Enemy scene 217

NavigationRegion2D node, creating 213-217

node groups 217

player, damaging in collision 220-222

script, writing 218-220

enemies and collectibles

base code, writing 225-227

entities, spawning automatically 227, 228

health potions, spawning 228, 229

scene structure, creating 222-225

spawning 222

enemies, Game Over screen

base scene, creating 229, 230

Game Over menu, showing when player dies 231-233

logic, adding to Game Over menu node 231

Enemy scene

NavigationAgent2D node, adding to 217

enumerations (enums) 299-301

versus strings 301

erase() function 68

Event Bus pattern 311

problem 311

solution 311-313

exported variables

types 136

export template

downloading 278, 279

F

FileAccess class

reference link 329

file paths 326

file system 325

flatcase 102

floating point 34

floating point numbers (floats) 34

floats 34

for loop 60, 61

free game assets 335

free open-source software (FOSS) 5

function 74, 105

defining 74, 75

naming 75

optional parameters 77, 78

pass keyword 77

return keyword 76

scope 84

versus method 74

G

game

actual exports, creating 279-283

actual web export, creating 285, 286

exporting, for Windows, Mac, and Linux 278

exporting, to consoles 295

exporting, to mobile platforms 295

exporting, to web 284, 285

multiplayer, removing 285

Object Pooling pattern, implementing 316-318

uploading, to Itch.io 283, 289-294

web export, zipping 286-289

game design document (GDD) 118

genre 118

mechanics 118
story 119
game ideas
card game 335
endless runner 335
platformer 335
puzzle game 335
turn-based strategy game 335
visual novel 335
game jams 284, 339
Game Maker's Toolkit
URL 336
GDQuest
URL 336
GD Quest's open RPG
URL 337
GDScript
constants 36
getter functions 133
Global Game Jam
URL 340
Godot 117

node-based system 118

values, changing on game run 136

Godot community 22

URL 22

Godot developers on Twitch

URL 338

Godot Engine

tutorials 336

used, for computer networking 251

Godot Engine demo projects

URL 337

Godot Engine Documentation 22, 337

URL 22, 336

Godot Engine project

contributing to 338, 339

Godot Engine Reddit

URL 338

Godot game engine 4

downloading 5-8

download link 8

history 4

light mode 10-13

main scene, creating 13-15

obtaining 5

open-source software 5

preparing 5

project, creating 8-10

script, writing 16-22

UI overview 15, 16

Godot Wild Jam

URL 340

H

has() function 68

Hearbeast

URL 336

highscore

storing, in autoloads 239

using, in main menu 243, 244

HighscoreManager autoload

creating 239-241

UI, adding in main menu and game scene 242, 243

I

if-else statement 41, 42

if statement 39-41

imperative programming language 91

implicit conversion 87

implicit type conversion 34

index 55

infinite loop 63

inheritance 80, 92

input events 160

input map tool 160

integers 34

interpreter 27

iOS

game, exporting to 295

IP address 252

Itch.io 277, 283, 284, 341

game, uploading to 289-294

URL 335

item 60

J

JavaScript Object Notation (JSON) 328

K

kebab-case 102

keep it simple, stupid (KISS) 105

Kenney

URL 335

key-value pair 66

creating 68

keywords 27

KidsCanCode

URL 336

L

lambda functions 301

creating 301, 302

reference link 303

using, cases 302

linear interpolation 184

little world, creating for player character 137

background color, creating 137, 138

creativity 142

node drawing order 139

outer wall, creating 140, 141

Polygon2D boulders, adding 138, 139

local area network (LAN) 247

Lone Wolf Technology 295

loops 60

break keyword 65, 66

continue keyword 64

for loop 60, 61

while loop 62, 63

M

magic numbers 37

magnitude 148

match statement 48, 49

menu

basic start menu, creating 206-210

control nodes 204

creating 203, 204

main scene, setting 210, 211

menu, control nodes

nodes, containing other nodes 205, 206

nodes, showing information 204, 205

nodes, taking input 205

metadata 326

method 74

versus function 74

method overriding 96, 97

multiplayer authority 261

multiplayer scene

enemy and collectibles, synchronizing 266, 267

EntitySpawner, synchronizing 265

MultiplayerSpawner, using to spawn player scenes 258-261

player code, updating 261, 262

player positions and health, synchronizing 262-264

projectile, synchronizing 267-269

updating 257

MultiplayerSpawner 258

multiple computers

connecting, from another computer 273, 274

server IP address, displaying 272, 273

used, for running game 271

N

naming conventions 102

PascalCase 102

SCREAMING_SNAKE_CASE 102

snake_case 102

naming tips 102

consistent 104

filler words, avoiding 103

meaningful and descriptive names 102, 103

names pronounceable 104

NavigationAgent2D node

adding, to Enemy scene 217

NavigationRegion2D node

creating 213-217

nested loops 69, 70

Nintendo 295

node group 217

null 70, 71

O

object-oriented (OO) 91

object-oriented programming (OOP) 91, 104

abstraction 93, 94

encapsulation 94

inheritance 92

polymorphism 94

object polymorphism 95, 96

Object Pooling pattern 314

implementing, in game 316-318

problem 314

solution 314-316

Observer pattern

reference link 313

official Godot Engine Discord

URL 338

official Godot Engine Forum

URL 338

old code

rewriting 310

OpenGameArt

URL 335

open-source software 4, 5

Operating System (OS) 278

optional parameters 77, 78

P

parameters 75

passing, by reference 304, 305

passing, by value 303, 304

Pascal case 80, 102

pass keyword 77

peer-to-peer network 250, 251

physical layer 248

physics engine 145

Pineapple Works 295

pixel art assets 123

player character

creating 119, 120

current player node, changing 156-158

health, displaying 124, 125

health label, updating with setters and getters 133-135

input actions 163

input mapping 159-161

moving 156

nodes, manipulating in editor 126, 127

physical forces, applying 158

physics process () function 159

player movement 163-165

process () function 159

sprite, adding 120-124

player node

collision shape, adding to 187-189

player script

creating 128, 129

node references, caching 130, 131

nodes, referencing 129, 130

testing 131

PlayStation 295

polymorphism 94

method overriding 96, 97

object polymorphism 95, 96

pool 314

port 252

port numbers

using 252, 253

private class members 104

programming patterns 310, 311

parts 310

programming style guides 106

blank line 108

line length 109

white space 107

project

tips, for starting 333, 334

projectiles

base scene, creating 233-235

logic, writing of 235, 236

shooting 233

spawning 236-239

protocol 249

public class members 104

R

range function 61, 62

remote tree 168, 169

autoloads 241

Remote Tree 318

return keyword 76

RigidBody2D 185, 186

running game

breakpoint 166-168

debugging 166

remote tree 168, 169

S

saved scenes

using 175, 176

save file 331

save manager

game, adjusting to use 330

preparing, for use in game 330

save system

creating 327

scalar 149

scaling vector 149

scancode 162

scene

branch, saving as 172

creating 37, 38

files, organizing 177, 178

root node 175

separate player scene, creating 172-174

scope 83

screaming snake case 36, 102

separate player scene

creating 172-174

setter functions 133

signals 194

singletons 163

snake case 27, 102

software architects 309

sprite 121

State Machines 319

example state 322, 323

problem 319

solution 319-322

static bodies

creating, for boulders 189

creating, for walls 190, 191

StaticBody2D 186

static function 299

static variables 299

Steam 277

strings 35

versus enumerations (enums) 301

working, in arrays 57

super keyword 298, 299

Super Mario Bros 319

survivor-like game

extending 334

T

ternary-if statement 50

timer

fixing 269

synchronizing 269

Transmission Control Protocol (TCP) 249

transport layer 248, 249

type hinting 85

arrays 86

autocompletion 89

editor 90, 91

null type 89

parameter function 86, 87

performance 90

return function 87

type inferring 88

using, for named classes 90

variable 85

Variant type 86

void, using as return function 88

type inferring 88

U

UI

adding, in main menu and game scene 242, 243

unit vector 155

User Datagram Protocol (UDP) 249

user path 326, 327

V

values

accessing, in arrays 55

accessing, in dictionaries 67

modifying, in dictionaries 67

vampire survivor-likes (VS) 118

variables 26

assignment operators 33, 34

exporting, to editor 132, 133

in GDScript 28

mathematical operators 32, 33

naming 27

naming, with correct characters 27

naming, with descriptive names 28, 29

printing out 29, 30

using, as drawers in filing cabinet 26, 27

value, changing 30, 31

Variant type 86

vector 147, 148

adding 149-151

clamping 152, 153

length 151, 152

normalizing 154, 155

rotating 153, 154

subtracting 149-151

Vector2

reference link 151

vector math refresher 146

2D coordinate system 146, 147

vector operations 151

distance_to() function 153

vector clamping 152, 153

vector length 151, 152

vector normalizing 154, 155

vector rotating 153, 154

W

while loop 62, 63

whitespace indentation 43

X

Xbox 295



packtpub.com

Подпишитесь на нашу онлайн-цифровую библиотеку для полного доступа к более чем 7000 книг и видео, а также к ведущим в отрасли инструментам, которые помогут вам спланировать свое личное развитие и продвинуться по карьерной лестнице. Для получения дополнительной информации посетите наш веб-сайт.

Зачем подписываться?

- Тратьте меньше времени на обучение и больше на программирование с помощью практических электронных книг и видео от более чем 4000 профессионалов отрасли
- Улучшите свое обучение с помощью планов развития навыков, созданных специально для вас
- Получайте бесплатную электронную книгу или видео каждый месяц
- Полная возможность поиска для легкого доступа к важной информации
- Копировать и вставлять, печатать и добавлять в закладки контент

Знаете ли вы, что Packt предлагает электронные версии каждой опубликованной книги, с доступными файлами PDF и ePub? Вы можете перейти на электронную версию packtpub.com, и как покупатель печатной книги вы имеете право на скидку на электронную копию. Свяжитесь с нами по адресу customercare@packtpub.com для получения более подробной информации.

На сайте www.packtpub.com вы также можете прочитать

коллекцию бесплатных технических статей, подписаться на ряд бесплатных информационных бюллетеней и получать эксклюзивные скидки и предложения на книги и электронные книги Packt.

Другие книги, которые вам могут понравиться

Если вам понравилась эта книга, вас могут заинтересовать и другие книги Packt:



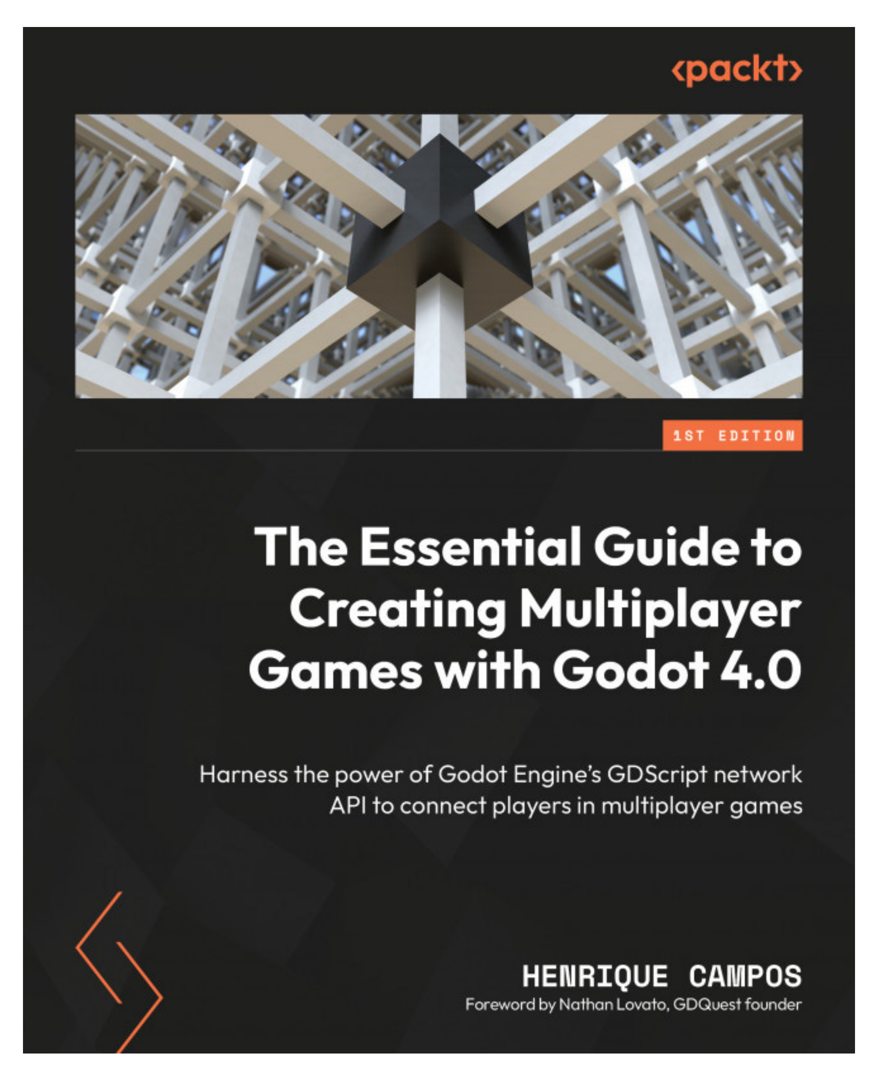
Проекты по разработке игр в Godot 4 (Godot 4 Game Development Projects)

Крис Брэдфилд (Chris Bradfield)

ISBN: 978-1-80461-040-4

- Познакомьтесь с игровым движком и редактором Godot, если вы новичок
- Изучите новые возможности Godot 4.0

- Создавайте игры в 2D и 3D, используя лучшие практики дизайна и кодирования
- Используйте систему узлов и сцен Godot для разработки надёжных, многоразовых игровых объектов
- Используйте GDScript, встроенный язык сценариев Godot, для создания сложных игровых систем
- Реализовать пользовательские интерфейсы для отображения информации
- Создавайте визуальные эффекты, чтобы разнообразить вашу игру
- Опубликуйте свою игру на настольных компьютерах и мобильных платформах



Основное руководство по созданию многопользовательских игр с помощью Godot 4.0 (The Essential Guide to Creating Multiplayer Games with Godot 4.)0

Энрике Кампос (Henrique Campos)

ISBN: 978-1-80323-261-4

- Понимать основы сетевых технологий и удаленного обмена данными между компьютерами

- Используйте встроенный API игрового движка Godot для настройки сети для игроков
- Освойте удаленные вызовы процедур и научитесь выполнять удаленные вызовы функций объектов
- Повысьте свои знания GDScript, чтобы максимально эффективно использовать этот мощный язык
- Изучите стандартные решения для распространенных многопользовательских онлайн-задач
- Улучшите свои компьютерных сетей и узнайте, как превратить однопользовательские игры в многопользовательские

Packt ищет авторов, похожих на вас

Если вы заинтересованы в том, чтобы стать автором для Packt, посетите authors.packtpub.com и подайте заявку сегодня. Мы работали с тысячами разработчиков и технических специалистов, таких как вы, чтобы помочь им поделиться своими идеями с мировым техническим сообществом. Вы можете подать общую заявку, подать заявку на определенную горячую тему, для которой мы набираем автора, или представить свою собственную идею.

Поделитесь своими мыслями

Теперь, когда вы закончили *изучать GDScript, разработав игру в Godot 4*, мы будем рады услышать ваши мысли! Если вы приобрели книгу на Amazon, нажмите здесь, чтобы перейти на страницу обзора Amazon для этой книги и поделиться своим отзывом или оставить отзыв на сайте, где вы её приобрели.

Ваш отзыв важен для нас и технического сообщества и поможет нам гарантировать предоставление контента высочайшего качества.

Загрузите бесплатную копию этой книги в формате PDF

Спасибо за покупку этой книги!

Вы любите читать на ходу, но не можете везде носить с собой печатные книги?

Приобретенная вами электронная книга несовместима с выбранным вами устройством?

Не волнуйтесь, теперь с каждой книгой Packt вы бесплатно получаете PDF-версию этой книги без DRM.

Читайте где угодно, в любом месте, на любом устройстве. Ищите, копируйте и вставляйте код из ваших любимых технических книг прямо в ваше приложение.

На этом преимущества не заканчиваются: вы можете получать эксклюзивный доступ к скидкам, информационным бюллетеням и отличному бесплатному контенту в своей электронной почте ежедневно.

Чтобы получить преимущества, выполните следующие простые шаги:

1. Отсканируйте QR-код или перейдите по ссылке ниже



https://packt.link/free-ebook/978-1-80461-698-7

1. Предоставьте доказательство покупки
2. Вот и всё! Мы отправим вам бесплатный PDF и другие преимущества на вашу электронную почту